Network Working Group
Request for Comments: 2578
STD: 58
Obsoletes: 1902
Category: Standards Track

Editors of this version:
K. McCloghrie
Cisco Systems
D. Perkins
SNMPinfo
J. Schoenwaelder
TU Braunschweig
Authors of previous version:
J. Case
SNMP Research
K. McCloghrie
Cisco Systems
M. Rose
First Virtual Holdings
S. Waldbusser
International Network Services
April 1999

Structure of Management Information Version 2 (SMIv2)

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction

Management information is viewed as a collection of managed objects, residing in a virtual information store, termed the Management Information Base (MIB). Collections of related objects are defined in MIB modules. These modules are written using an adapted subset of OSI's Abstract Syntax Notation One, ASN.1 (1988) [1]. It is the purpose of this document, the Structure of Management Information (SMI), to define that adapted subset, and to assign a set of associated administrative values.

The SMI is divided into three parts: module definitions, object definitions, and, notification definitions.

(1) Module definitions are used when describing information modules. An ASN.1 macro, MODULE-IDENTITY, is used to concisely convey the semantics of an information module.

(2) Object definitions are used when describing managed objects. An ASN.1 macro, OBJECT-TYPE, is used to concisely convey the syntax and semantics of a managed object.

(3) Notification definitions are used when describing unsolicited transmissions of management information. An ASN.1 macro, NOTIFICATION-TYPE, is used to concisely convey the syntax and semantics of a notification.

1.1. A Note on Terminology

For the purpose of exposition, the original Structure of Management Information, as described in RFCs 1155 (STD 16), 1212 (STD 16), and RFC 1215, is termed the SMI version 1 (SMIv1). The current version of the Structure of Management Information is termed SMI version 2 (SMIv2).

2. Definitions

```
SNMPv2-SMI DEFINITIONS ::= BEGIN


-- the path to the root

org             OBJECT IDENTIFIER ::= { iso 3 }  --  "iso" = 1
dod             OBJECT IDENTIFIER ::= { org 6 }
internet        OBJECT IDENTIFIER ::= { dod 1 }

directory       OBJECT IDENTIFIER ::= { internet 1 }

mgmt            OBJECT IDENTIFIER ::= { internet 2 }
mib-2           OBJECT IDENTIFIER ::= { mgmt 1 }
transmission    OBJECT IDENTIFIER ::= { mib-2 10 }

experimental    OBJECT IDENTIFIER ::= { internet 3 }

private         OBJECT IDENTIFIER ::= { internet 4 }
enterprises     OBJECT IDENTIFIER ::= { private 1 }

security        OBJECT IDENTIFIER ::= { internet 5 }

snmpV2          OBJECT IDENTIFIER ::= { internet 6 }

-- transport domains
snmpDomains     OBJECT IDENTIFIER ::= { snmpV2 1 }

-- transport proxies
snmpProxys      OBJECT IDENTIFIER ::= { snmpV2 2 }

-- module identities
snmpModules     OBJECT IDENTIFIER ::= { snmpV2 3 }

-- Extended UTCTime, to allow dates with four-digit years
-- (Note that this definition of ExtUTCTime is not to be IMPORTed
--  by MIB modules.)
ExtUTCTime ::= OCTET STRING(SIZE(11 | 13))
    -- format is YYMMDDHHMMZ or YYYYMMDDHHMMZ
```

```
    --   where: YY   - last two digits of year (only years
    --                 between 1900-1999)
    --          YYYY - last four digits of the year (any year)
    --          MM   - month (01 through 12)
    --          DD   - day of month (01 through 31)
    --          HH   - hours (00 through 23)
    --          MM   - minutes (00 through 59)
    --          Z    - denotes GMT (the ASCII character Z)
    --
    -- For example, "9502192015Z" and "199502192015Z" represent
    -- 8:15pm GMT on 19 February 1995. Years after 1999 must use
    -- the four digit year format. Years 1900-1999 may use the
    -- two or four digit format.

-- definitions for information modules

MODULE-IDENTITY MACRO ::=
BEGIN
    TYPE NOTATION ::=
                  "LAST-UPDATED" value(Update ExtUTCTime)
                  "ORGANIZATION" Text
                  "CONTACT-INFO" Text
                  "DESCRIPTION" Text
                  RevisionPart

    VALUE NOTATION ::=
                  value(VALUE OBJECT IDENTIFIER)

    RevisionPart ::=
                  Revisions
                | empty
    Revisions ::=
                  Revision
                | Revisions Revision
    Revision ::=
                  "REVISION" value(Update ExtUTCTime)
                  "DESCRIPTION" Text

    -- a character string as defined in section 3.1.1
    Text ::= value(IA5String)
END


OBJECT-IDENTITY MACRO ::=
BEGIN
    TYPE NOTATION ::=
                  "STATUS" Status
                  "DESCRIPTION" Text
```

```
                ReferPart

    VALUE NOTATION ::=
                value(VALUE OBJECT IDENTIFIER)

    Status ::=
                "current"
              | "deprecated"
              | "obsolete"

    ReferPart ::=
                "REFERENCE" Text
              | empty

    -- a character string as defined in section 3.1.1
    Text ::= value(IA5String)
END


-- names of objects
-- (Note that these definitions of ObjectName and NotificationName
--  are not to be IMPORTed by MIB modules.)

ObjectName ::=
    OBJECT IDENTIFIER

NotificationName ::=
    OBJECT IDENTIFIER

-- syntax of objects

-- the "base types" defined here are:
--   3 built-in ASN.1 types: INTEGER, OCTET STRING, OBJECT IDENTIFIER
--   8 application-defined types: Integer32, IpAddress, Counter32,
--              Gauge32, Unsigned32, TimeTicks, Opaque, and Counter64

ObjectSyntax ::=
    CHOICE {
        simple
            SimpleSyntax,

          -- note that SEQUENCEs for conceptual tables and
          -- rows are not mentioned here...

        application-wide
            ApplicationSyntax
    }
```

```
-- built-in ASN.1 types

SimpleSyntax ::=
    CHOICE {
        -- INTEGERs with a more restrictive range
        -- may also be used
        integer-value               -- includes Integer32
            INTEGER (-2147483648..2147483647),

        -- OCTET STRINGs with a more restrictive size
        -- may also be used
        string-value
            OCTET STRING (SIZE (0..65535)),

        objectID-value
            OBJECT IDENTIFIER
    }

-- indistinguishable from INTEGER, but never needs more than
-- 32-bits for a two's complement representation
Integer32 ::=
        INTEGER (-2147483648..2147483647)


-- application-wide types

ApplicationSyntax ::=
    CHOICE {
        ipAddress-value
            IpAddress,

        counter-value
            Counter32,

        timeticks-value
            TimeTicks,

        arbitrary-value
            Opaque,

        big-counter-value
            Counter64,

        unsigned-integer-value  -- includes Gauge32
            Unsigned32
    }

-- in network-byte order
```

```
-- (this is a tagged type for historical reasons)
IpAddress ::=
    [APPLICATION 0]
        IMPLICIT OCTET STRING (SIZE (4))

-- this wraps
Counter32 ::=
    [APPLICATION 1]
        IMPLICIT INTEGER (0..4294967295)

-- this doesn't wrap
Gauge32 ::=
    [APPLICATION 2]
        IMPLICIT INTEGER (0..4294967295)

-- an unsigned 32-bit quantity
-- indistinguishable from Gauge32
Unsigned32 ::=
    [APPLICATION 2]
        IMPLICIT INTEGER (0..4294967295)

-- hundredths of seconds since an epoch
TimeTicks ::=
    [APPLICATION 3]
        IMPLICIT INTEGER (0..4294967295)

-- for backward-compatibility only
Opaque ::=
    [APPLICATION 4]
        IMPLICIT OCTET STRING

-- for counters that wrap in less than one hour with only 32 bits
Counter64 ::=
    [APPLICATION 6]
        IMPLICIT INTEGER (0..18446744073709551615)


-- definition for objects

OBJECT-TYPE MACRO ::=
BEGIN
    TYPE NOTATION ::=
                  "SYNTAX" Syntax
                  UnitsPart
                  "MAX-ACCESS" Access
                  "STATUS" Status
                  "DESCRIPTION" Text
                  ReferPart
```

```
                IndexPart
                DefValPart

    VALUE NOTATION ::=
                value(VALUE ObjectName)

    Syntax ::=   -- Must be one of the following:
                      -- a base type (or its refinement),
                      -- a textual convention (or its refinement), or
                      -- a BITS pseudo-type
                  type
               | "BITS" "{" NamedBits "}"

    NamedBits ::= NamedBit
               | NamedBits "," NamedBit

    NamedBit ::=  identifier "(" number ")" -- number is nonnegative

    UnitsPart ::=
                  "UNITS" Text
               | empty

    Access ::=
                  "not-accessible"
               | "accessible-for-notify"
               | "read-only"
               | "read-write"
               | "read-create"

    Status ::=
                  "current"
               | "deprecated"
               | "obsolete"

    ReferPart ::=
                  "REFERENCE" Text
               | empty

    IndexPart ::=
                  "INDEX"    "{" IndexTypes "}"
               | "AUGMENTS" "{" Entry      "}"
               | empty
    IndexTypes ::=
                  IndexType
               | IndexTypes "," IndexType
    IndexType ::=
                  "IMPLIED" Index
               | Index
```

```
    Index ::=
                    -- use the SYNTAX value of the
                    -- correspondent OBJECT-TYPE invocation
                  value(ObjectName)
    Entry ::=
                    -- use the INDEX value of the
                    -- correspondent OBJECT-TYPE invocation
                  value(ObjectName)

    DefValPart ::= "DEFVAL" "{" Defvalue "}"
                | empty

    Defvalue ::=  -- must be valid for the type specified in
                  -- SYNTAX clause of same OBJECT-TYPE macro
                  value(ObjectSyntax)
                | "{" BitsValue "}"

    BitsValue ::= BitNames
                | empty

    BitNames ::=  BitName
                | BitNames "," BitName

    BitName ::= identifier

    -- a character string as defined in section 3.1.1
    Text ::= value(IA5String)
END


-- definitions for notifications

NOTIFICATION-TYPE MACRO ::=
BEGIN
    TYPE NOTATION ::=
                  ObjectsPart
                  "STATUS" Status
                  "DESCRIPTION" Text
                  ReferPart

    VALUE NOTATION ::=
                  value(VALUE NotificationName)

    ObjectsPart ::=
                  "OBJECTS" "{" Objects "}"
                | empty
    Objects ::=
                  Object
```

```
                | Objects "," Object
    Object ::=
                  value(ObjectName)

    Status ::=
                  "current"
                | "deprecated"
                | "obsolete"

    ReferPart ::=
                  "REFERENCE" Text
                | empty

    -- a character string as defined in section 3.1.1
    Text ::= value(IA5String)
END

-- definitions of administrative identifiers

zeroDotZero    OBJECT-IDENTITY
    STATUS     current
    DESCRIPTION
            "A value used for null identifiers."
    ::= { 0 0 }

END
```

3. Information Modules

An "information module" is an ASN.1 module defining information relating to network management.

The SMI describes how to use an adapted subset of ASN.1 (1988) to define an information module. Further, additional restrictions are placed on "standard" information modules. It is strongly recommended that "enterprise-specific" information modules also adhere to these restrictions.

Typically, there are three kinds of information modules:

(1) MIB modules, which contain definitions of inter-related managed objects, make use of the OBJECT-TYPE and NOTIFICATION-TYPE macros;

(2) compliance statements for MIB modules, which make use of the MODULE-COMPLIANCE and OBJECT-GROUP macros [2]; and,

(3) capability statements for agent implementations which make use of the AGENT-CAPABILITIES macros [2].

This classification scheme does not imply a rigid taxonomy. For example, a "standard" information module will normally include definitions of managed objects and a compliance statement. Similarly, an "enterprise-specific" information module might include definitions of managed objects and a capability statement. Of course, a "standard" information module may not contain capability statements.

The constructs of ASN.1 allowed in SMIv2 information modules include: the IMPORTS clause, value definitions for OBJECT IDENTIFIERs, type definitions for SEQUENCEs (with restrictions), ASN.1 type assignments of the restricted ASN.1 types allowed in SMIv2, and instances of ASN.1 macros defined in this document and its companion documents [2, 3]. Additional ASN.1 macros must not be defined in SMIv2 information modules. SMIv1 macros must not be used in SMIv2 information modules.

The names of all standard information modules must be unique (but different versions of the same information module should have the same name). Developers of enterprise information modules are encouraged to choose names for their information modules that will have a low probability of colliding with standard or other enterprise information modules. An information module may not use the ASN.1 construct of placing an object identifier value between the module name and the "DEFINITIONS" keyword. For the purposes of this specification, an ASN.1 module name begins with an upper-case letter and continues with zero or more letters, digits, or hyphens, except that a hyphen can not be the last character, nor can there be two consecutive hyphens.

All information modules start with exactly one invocation of the MODULE-IDENTITY macro, which provides contact information as well as revision history to distinguish between versions of the same information module. This invocation must appear immediately after any IMPORTs statements.

3.1. Macro Invocation

Within an information module, each macro invocation appears as:

```
<descriptor> <macro> <clauses> ::= <value>
```

where <descriptor> corresponds to an ASN.1 identifier, <macro> names the macro being invoked, and <clauses> and <value> depend on the definition of the macro. (Note that this definition of a descriptor applies to all macros defined in this memo and in [2].)

For the purposes of this specification, an ASN.1 identifier consists of one or more letters or digits, and its initial character must be a lower-case letter. Note that hyphens are not allowed by this specification (except for use by information modules converted from SMIv1 which did allow hyphens).

For all descriptors appearing in an information module, the descriptor shall be unique and mnemonic, and shall not exceed 64 characters in length. (However, descriptors longer than 32 characters are not recommended.) This promotes a common language for humans to use when discussing the information module and also facilitates simple table mappings for user-interfaces.

The set of descriptors defined in all "standard" information modules shall be unique.

Finally, by convention, if the descriptor refers to an object with a SYNTAX clause value of either Counter32 or Counter64, then the descriptor used for the object should denote plurality.

3.1.1. Textual Values and Strings

Some clauses in a macro invocation may take a character string as a textual value (e.g., the DESCRIPTION clause). Other clauses take binary or hexadecimal strings (in any position where a non-negative number is allowed).

A character string is preceded and followed by the quote character ("), and consists of an arbitrary number (possibly zero) of:

- any 7-bit displayable ASCII characters except quote ("),
- tab characters,
- spaces, and
- line terminator characters (\n or \r\n).

The value of a character string is interpreted as ASCII.

A binary string consists of a number (possibly zero) of zeros and ones preceded by a single (') and followed by either the pair ('B) or ('b), where the number is a multiple of eight.

A hexadecimal string consists of an even number (possibly zero) of hexadecimal digits, preceded by a single (') and followed by either the pair ('H) or ('h). Digits specified via letters can be in upper or lower case.

Note that ASN.1 comments can not be enclosed inside any of these types of strings.

3.2. IMPORTing Symbols

To reference an external object, the IMPORTS statement must be used to identify both the descriptor and the module in which the descriptor is defined, where the module is identified by its ASN.1 module name.

Note that when symbols from "enterprise-specific" information modules are referenced (e.g., a descriptor), there is the possibility of collision. As such, if different objects with the same descriptor are IMPORTed, then this ambiguity is resolved by prefixing the descriptor with the name of the information module and a dot ("."), i.e.,

"module.descriptor"

(All descriptors must be unique within any information module.)

Of course, this notation can be used to refer to objects even when there is no collision when IMPORTing symbols.

Finally, if any of the ASN.1 named types and macros defined in this document, specifically:

Counter32, Counter64, Gauge32, Integer32, IpAddress, MODULE-IDENTITY, NOTIFICATION-TYPE, Opaque, OBJECT-TYPE, OBJECT-IDENTITY, TimeTicks, Unsigned32,

or any of those defined in [2] or [3], are used in an information module, then they must be imported using the IMPORTS statement. However, the following must not be included in an IMPORTS statement:

- named types defined by ASN.1 itself, specifically: INTEGER, OCTET STRING, OBJECT IDENTIFIER, SEQUENCE, SEQUENCE OF type,
- the BITS construct.

3.3. Exporting Symbols

The ASN.1 EXPORTS statement is not allowed in SMIv2 information modules. All items defined in an information module are automatically exported.

3.4. ASN.1 Comments

ASN.1 comments can be included in an information module. However, it is recommended that all substantive descriptions be placed within an appropriate DESCRIPTION clause.

ASN.1 comments commence with a pair of adjacent hyphens and end with the next pair of adjacent hyphens or at the end of the line, whichever occurs first. Comments ended by a pair of hyphens have the effect of a single space character.

3.5. OBJECT IDENTIFIER values

An OBJECT IDENTIFIER value is an ordered list of non-negative numbers. For the SMIv2, each number in the list is referred to as a sub-identifier, there are at most 128 sub-identifiers in a value, and each sub-identifier has a maximum value of 2^32-1 (4294967295 decimal).

All OBJECT IDENTIFIER values have at least two sub-identifiers, where the value of the first sub-identifier is one of the following well-known names:

Value	Name
0	ccitt
1	iso
2	joint-iso-ccitt

(Note that this SMI does not recognize "new" well-known names, e.g., as defined when the CCITT became the ITU.)

3.6. OBJECT IDENTIFIER usage

OBJECT IDENTIFIERs are used in information modules in two ways:

(1) registration: the definition of a particular item is registered as a particular OBJECT IDENTIFIER value, and associated with a particular descriptor. After such a registration, the semantics thereby associated with the value are not allowed to change, the OBJECT IDENTIFIER can not be used for any other registration, and the descriptor can not be changed nor associated with any other registration. The following macros result in a registration:

    OBJECT-TYPE, MODULE-IDENTITY, NOTIFICATION-TYPE, OBJECT-GROUP, OBJECT-IDENTITY, NOTIFICATION-GROUP, MODULE-COMPLIANCE, AGENT-CAPABILITIES.

(2) assignment: a descriptor can be assigned to a particular OBJECT IDENTIFIER value. For this usage, the semantics associated with the OBJECT IDENTIFIER value is not allowed to change, and a descriptor assigned to a particular OBJECT IDENTIFIER value cannot subsequently be assigned to another. However, multiple descriptors can be assigned to the same OBJECT IDENTIFIER value. Such assignments are specified in the following manner:

```
mib         OBJECT IDENTIFIER ::= { mgmt 1 }  -- from RFC1156
mib-2       OBJECT IDENTIFIER ::= { mgmt 1 }  -- from RFC1213
fredRouter  OBJECT IDENTIFIER ::= { flintStones 1 1 }
barneySwitch OBJECT IDENTIFIER ::= { flintStones bedrock(2) 1 }
```

Note while the above examples are legal, the following is not:

```
dinoHost OBJECT IDENTIFIER ::= { flintStones bedrock 2 }
```

A descriptor is allowed to be associated with both a registration and an assignment, providing both are associated with the same OBJECT IDENTIFIER value and semantics.

3.7. Reserved Keywords

The following are reserved keywords which must not be used as descriptors or module names:

```
ABSENT ACCESS AGENT-CAPABILITIES ANY APPLICATION AUGMENTS BEGIN
BIT BITS BOOLEAN BY CHOICE COMPONENT COMPONENTS CONTACT-INFO
CREATION-REQUIRES Counter32 Counter64 DEFAULT DEFINED
DEFINITIONS DEFVAL DESCRIPTION DISPLAY-HINT END ENUMERATED
ENTERPRISE EXPLICIT EXPORTS EXTERNAL FALSE FROM GROUP Gauge32
IDENTIFIER IMPLICIT IMPLIED IMPORTS INCLUDES INDEX INTEGER
Integer32 IpAddress LAST-UPDATED MANDATORY-GROUPS MAX MAX-ACCESS
MIN MIN-ACCESS MINUS-INFINITY MODULE MODULE-COMPLIANCE MODULE-
IDENTITY NOTIFICATION-GROUP NOTIFICATION-TYPE NOTIFICATIONS NULL
OBJECT OBJECT-GROUP OBJECT-IDENTITY OBJECT-TYPE OBJECTS OCTET OF
OPTIONAL ORGANIZATION Opaque PLUS-INFINITY PRESENT PRIVATE
PRODUCT-RELEASE REAL REFERENCE REVISION SEQUENCE SET SIZE STATUS
STRING SUPPORTS SYNTAX TAGS TEXTUAL-CONVENTION TRAP-TYPE TRUE
TimeTicks UNITS UNIVERSAL Unsigned32 VARIABLES VARIATION WITH
WRITE-SYNTAX
```

4. Naming Hierarchy

The root of the subtree administered by the Internet Assigned Numbers Authority (IANA) for the Internet is:

```
internet       OBJECT IDENTIFIER ::= { iso 3 6 1 }
```

That is, the Internet subtree of OBJECT IDENTIFIERs starts with the prefix:

```
1.3.6.1.
```

Several branches underneath this subtree are used for network management:

```
mgmt          OBJECT IDENTIFIER ::= { internet 2 }
experimental  OBJECT IDENTIFIER ::= { internet 3 }
private       OBJECT IDENTIFIER ::= { internet 4 }
enterprises   OBJECT IDENTIFIER ::= { private 1 }
```

However, the SMI does not prohibit the definition of objects in other portions of the object tree.

The mgmt(2) subtree is used to identify "standard" objects.

The experimental(3) subtree is used to identify objects being designed by working groups of the IETF. If an information module produced by a working group becomes a "standard" information module, then at the very beginning of its entry onto the Internet standards track, the objects are moved under the mgmt(2) subtree.

The private(4) subtree is used to identify objects defined unilaterally. The enterprises(1) subtree beneath private is used, among other things, to permit providers of networking subsystems to register models of their products.

5. Mapping of the MODULE-IDENTITY macro

The MODULE-IDENTITY macro is used to provide contact and revision history for each information module. It must appear exactly once in every information module. It should be noted that the expansion of the MODULE-IDENTITY macro is something which conceptually happens during implementation and not during run-time.

Note that reference in an IMPORTS clause or in clauses of SMIv2 macros to an information module is NOT through the use of the 'descriptor' of a MODULE-IDENTITY macro; rather, an information module is referenced through specifying its module name.

5.1. Mapping of the LAST-UPDATED clause

The LAST-UPDATED clause, which must be present, contains the date and time that this information module was last edited.

5.2. Mapping of the ORGANIZATION clause

The ORGANIZATION clause, which must be present, contains a textual description of the organization under whose auspices this information module was developed.

5.3. Mapping of the CONTACT-INFO clause

The CONTACT-INFO clause, which must be present, contains the name, postal address, telephone number, and electronic mail address of the person to whom technical queries concerning this information module should be sent.

5.4. Mapping of the DESCRIPTION clause

The DESCRIPTION clause, which must be present, contains a high-level textual description of the contents of this information module.

5.5. Mapping of the REVISION clause

The REVISION clause, which need not be present, is repeatedly used to describe the revisions (including the initial version) made to this information module, in reverse chronological order (i.e., most recent first). Each instance of this clause contains the date and time of the revision.

5.5.1. Mapping of the DESCRIPTION sub-clause

The DESCRIPTION sub-clause, which must be present for each REVISION clause, contains a high-level textual description of the revision identified in that REVISION clause.

5.6. Mapping of the MODULE-IDENTITY value

The value of an invocation of the MODULE-IDENTITY macro is an OBJECT IDENTIFIER. As such, this value may be authoritatively used when specifying an OBJECT IDENTIFIER value to refer to the information module containing the invocation.

Note that it is a common practice to use the value of the MODULE-IDENTITY macro as a subtree under which other OBJECT IDENTIFIER values assigned within the module are defined. However, it is legal (and occasionally necessary) for the other OBJECT IDENTIFIER values assigned within the module to be unrelated to the OBJECT IDENTIFIER value of the MODULE-IDENTITY macro.

5.7. Usage Example

Consider how a skeletal MIB module might be constructed: e.g.,

```
FIZBIN-MIB DEFINITIONS ::= BEGIN

IMPORTS
    MODULE-IDENTITY, OBJECT-TYPE, experimental
```

```
          FROM SNMPv2-SMI;


   fizbin MODULE-IDENTITY
       LAST-UPDATED "199505241811Z"
       ORGANIZATION "IETF SNMPv2 Working Group"
       CONTACT-INFO
               "        Marshall T. Rose

                Postal: Dover Beach Consulting, Inc.
                        420 Whisman Court
                        Mountain View, CA  94043-2186
                        US

                   Tel: +1 415 968 1052
                   Fax: +1 415 968 2510

                E-mail: mrose@dbc.mtview.ca.us"

       DESCRIPTION
               "The MIB module for entities implementing the xxxx
               protocol."
       REVISION      "9505241811Z"
       DESCRIPTION
               "The latest version of this MIB module."
       REVISION      "9210070433Z"
       DESCRIPTION
               "The initial version of this MIB module, published in
               RFC yyyy."
   -- contact IANA for actual number
       ::= { experimental xx }

   END
```

6. Mapping of the OBJECT-IDENTITY macro

The OBJECT-IDENTITY macro is used to define information about an OBJECT IDENTIFIER assignment. All administrative OBJECT IDENTIFIER assignments which define a type identification value (see AutonomousType, a textual convention defined in [3]) should be defined via the OBJECT-IDENTITY macro. It should be noted that the expansion of the OBJECT-IDENTITY macro is something which conceptually happens during implementation and not during run-time.

6.1. Mapping of the STATUS clause

The STATUS clause, which must be present, indicates whether this definition is current or historic.

The value "current" means that the definition is current and valid. The value "obsolete" means the definition is obsolete and should not be implemented and/or can be removed if previously implemented. While the value "deprecated" also indicates an obsolete definition, it permits new/continued implementation in order to foster interoperability with older/existing implementations.

6.2. Mapping of the DESCRIPTION clause

The DESCRIPTION clause, which must be present, contains a textual description of the object assignment.

6.3. Mapping of the REFERENCE clause

The REFERENCE clause, which need not be present, contains a textual cross-reference to some other document, either another information module which defines a related assignment, or some other document which provides additional information relevant to this definition.

6.4. Mapping of the OBJECT-IDENTITY value

The value of an invocation of the OBJECT-IDENTITY macro is an OBJECT IDENTIFIER.

6.5. Usage Example

Consider how an OBJECT IDENTIFIER assignment might be made: e.g.,

```
fizbin69 OBJECT-IDENTITY
    STATUS  current
    DESCRIPTION
            "The authoritative identity of the Fizbin 69 chipset."
   ::= { fizbinChipSets 1 }
```

7. Mapping of the OBJECT-TYPE macro

The OBJECT-TYPE macro is used to define a type of managed object. It should be noted that the expansion of the OBJECT-TYPE macro is something which conceptually happens during implementation and not during run-time.

For leaf objects which are not columnar objects (i.e., not contained within a conceptual table), instances of the object are identified by appending a sub-identifier of zero to the name of that object. Otherwise, the INDEX clause of the conceptual row object superior to a columnar object defines instance identification information.

7.1. Mapping of the SYNTAX clause

The SYNTAX clause, which must be present, defines the abstract data structure corresponding to that object. The data structure must be one of the following: a base type, the BITS construct, or a textual convention. (SEQUENCE OF and SEQUENCE are also possible for conceptual tables, see section 7.1.12). The base types are those defined in the ObjectSyntax CHOICE. A textual convention is a newly-defined type defined as a sub-type of a base type [3].

An extended subset of the full capabilities of ASN.1 (1988) sub-typing is allowed, as appropriate to the underlying ASN.1 type. Any such restriction on size, range or enumerations specified in this clause represents the maximal level of support which makes "protocol sense". Restrictions on sub-typing are specified in detail in Section 9 and Appendix A of this memo.

The semantics of ObjectSyntax are now described.

7.1.1. Integer32 and INTEGER

The Integer32 type represents integer-valued information between -2^31 and 2^31-1 inclusive (-2147483648 to 2147483647 decimal). This type is indistinguishable from the INTEGER type. Both the INTEGER and Integer32 types may be sub-typed to be more constrained than the Integer32 type.

The INTEGER type (but not the Integer32 type) may also be used to represent integer-valued information as named-number enumerations. In this case, only those named-numbers so enumerated may be present as a value. Note that although it is recommended that enumerated values start at 1 and be numbered contiguously, any valid value for Integer32 is allowed for an enumerated value and, further, enumerated values needn’t be contiguously assigned.

Finally, a label for a named-number enumeration must consist of one or more letters or digits, up to a maximum of 64 characters, and the initial character must be a lower-case letter. (However, labels longer than 32 characters are not recommended.) Note that hyphens are not allowed by this specification (except for use by information modules converted from SMIv1 which did allow hyphens).

7.1.2. OCTET STRING

The OCTET STRING type represents arbitrary binary or textual data. Although the SMI-specified size limitation for this type is 65535 octets, MIB designers should realize that there may be implementation and interoperability limitations for sizes in excess of 255 octets.

7.1.3. OBJECT IDENTIFIER

The OBJECT IDENTIFIER type represents administratively assigned names. Any instance of this type may have at most 128 sub-identifiers. Further, each sub-identifier must not exceed the value 2^32-1 (4294967295 decimal).

7.1.4. The BITS construct

The BITS construct represents an enumeration of named bits. This collection is assigned non-negative, contiguous (but see below) values, starting at zero. Only those named-bits so enumerated may be present in a value. (Thus, enumerations must be assigned to consecutive bits; however, see Section 9 for refinements of an object with this syntax.)

As part of updating an information module, for an object defined using the BITS construct, new enumerations can be added or existing enumerations can have new labels assigned to them. After an enumeration is added, it might not be possible to distinguish between an implementation of the updated object for which the new enumeration is not asserted, and an implementation of the object prior to the addition. Depending on the circumstances, such an ambiguity could either be desirable or could be undesirable. The means to avoid such an ambiguity is dependent on the encoding of values on the wire; however, one possibility is to define new enumerations starting at the next multiple of eight bits. (Of course, this can also result in the enumerations no longer being contiguous.)

Although there is no SMI-specified limitation on the number of enumerations (and therefore on the length of a value), except as may be imposed by the limit on the length of an OCTET STRING, MIB designers should realize that there may be implementation and interoperability limitations for sizes in excess of 128 bits.

Finally, a label for a named-number enumeration must consist of one or more letters or digits, up to a maximum of 64 characters, and the initial character must be a lower-case letter. (However, labels longer than 32 characters are not recommended.) Note that hyphens are not allowed by this specification.

7.1.5. IpAddress

The IpAddress type represents a 32-bit internet address. It is represented as an OCTET STRING of length 4, in network byte-order.

Note that the IpAddress type is a tagged type for historical reasons. Network addresses should be represented using an invocation of the TEXTUAL-CONVENTION macro [3].

7.1.6. Counter32

The Counter32 type represents a non-negative integer which monotonically increases until it reaches a maximum value of 2^32-1 (4294967295 decimal), when it wraps around and starts increasing again from zero.

Counters have no defined "initial" value, and thus, a single value of a Counter has (in general) no information content. Discontinuities in the monotonically increasing value normally occur at re-initialization of the management system, and at other times as specified in the description of an object-type using this ASN.1 type. If such other times can occur, for example, the creation of an object instance at times other than re-initialization, then a corresponding object should be defined, with an appropriate SYNTAX clause, to indicate the last discontinuity. Examples of appropriate SYNTAX clause include: TimeStamp (a textual convention defined in [3]), DateAndTime (another textual convention from [3]) or TimeTicks.

The value of the MAX-ACCESS clause for objects with a SYNTAX clause value of Counter32 is either "read-only" or "accessible-for-notify".

A DEFVAL clause is not allowed for objects with a SYNTAX clause value of Counter32.

7.1.7. Gauge32

The Gauge32 type represents a non-negative integer, which may increase or decrease, but shall never exceed a maximum value, nor fall below a minimum value. The maximum value can not be greater than 2^32-1 (4294967295 decimal), and the minimum value can not be smaller than 0. The value of a Gauge32 has its maximum value whenever the information being modeled is greater than or equal to its maximum value, and has its minimum value whenever the information being modeled is smaller than or equal to its minimum value. If the information being modeled subsequently decreases below (increases above) the maximum (minimum) value, the Gauge32 also decreases (increases). (Note that despite of the use of the term "latched" in the original definition of this type, it does not become "stuck" at its maximum or minimum value.)

7.1.8. TimeTicks

The TimeTicks type represents a non-negative integer which represents the time, modulo 2^32 (4294967296 decimal), in hundredths of a second between two epochs. When objects are defined which use this ASN.1 type, the description of the object identifies both of the reference epochs.

For example, [3] defines the TimeStamp textual convention which is based on the TimeTicks type. With a TimeStamp, the first reference epoch is defined as the time when sysUpTime [5] was zero, and the second reference epoch is defined as the current value of sysUpTime.

The TimeTicks type may not be sub-typed.

7.1.9. Opaque

The Opaque type is provided solely for backward-compatibility, and shall not be used for newly-defined object types.

The Opaque type supports the capability to pass arbitrary ASN.1 syntax. A value is encoded using the ASN.1 Basic Encoding Rules [4] into a string of octets. This, in turn, is encoded as an OCTET STRING, in effect "double-wrapping" the original ASN.1 value.

Note that a conforming implementation need only be able to accept and recognize opaquely-encoded data. It need not be able to unwrap the data and then interpret its contents.

A requirement on "standard" MIB modules is that no object may have a SYNTAX clause value of Opaque.

7.1.10. Counter64

The Counter64 type represents a non-negative integer which monotonically increases until it reaches a maximum value of 2^64-1 (18446744073709551615 decimal), when it wraps around and starts increasing again from zero.

Counters have no defined "initial" value, and thus, a single value of a Counter has (in general) no information content. Discontinuities in the monotonically increasing value normally occur at re-initialization of the management system, and at other times as specified in the description of an object-type using this ASN.1 type. If such other times can occur, for example, the creation of an object instance at times other than re-initialization, then a corresponding object should be defined, with an appropriate SYNTAX clause, to indicate the last discontinuity. Examples of appropriate SYNTAX

clause are: TimeStamp (a textual convention defined in [3]), DateAndTime (another textual convention from [3]) or TimeTicks.

The value of the MAX-ACCESS clause for objects with a SYNTAX clause value of Counter64 is either "read-only" or "accessible-for-notify".

A requirement on "standard" MIB modules is that the Counter64 type may be used only if the information being modeled would wrap in less than one hour if the Counter32 type was used instead.

A DEFVAL clause is not allowed for objects with a SYNTAX clause value of Counter64.

7.1.11. Unsigned32

The Unsigned32 type represents integer-valued information between 0 and 2^32-1 inclusive (0 to 4294967295 decimal).

7.1.12. Conceptual Tables

Management operations apply exclusively to scalar objects. However, it is sometimes convenient for developers of management applications to impose an imaginary, tabular structure on an ordered collection of objects within the MIB. Each such conceptual table contains zero or more rows, and each row may contain one or more scalar objects, termed columnar objects. This conceptualization is formalized by using the OBJECT-TYPE macro to define both an object which corresponds to a table and an object which corresponds to a row in that table. A conceptual table has SYNTAX of the form:

```
SEQUENCE OF <EntryType>
```

where <EntryType> refers to the SEQUENCE type of its subordinate conceptual row. A conceptual row has SYNTAX of the form:

```
<EntryType>
```

where <EntryType> is a SEQUENCE type defined as follows:

```
<EntryType> ::= SEQUENCE { <type1>, ... , <typeN> }
```

where there is one <type> for each subordinate object, and each <type> is of the form:

```
<descriptor> <syntax>
```

where <descriptor> is the descriptor naming a subordinate object, and <syntax> has the value of that subordinate object's SYNTAX clause,

except that both sub-typing information and the named values for enumerated integers or the named bits for the BITS construct, are omitted from <syntax>.

Further, a <type> is always present for every subordinate object. (The ASN.1 DEFAULT and OPTIONAL clauses are disallowed in the SEQUENCE definition.) The MAX-ACCESS clause for conceptual tables and rows is "not-accessible".

7.1.12.1. Creation and Deletion of Conceptual Rows

For newly-defined conceptual rows which allow the creation of new object instances and/or the deletion of existing object instances, there should be one columnar object with a SYNTAX clause value of RowStatus (a textual convention defined in [3]) and a MAX-ACCESS clause value of read-create. By convention, this is termed the status column for the conceptual row.

7.2. Mapping of the UNITS clause

This UNITS clause, which need not be present, contains a textual definition of the units associated with that object.

7.3. Mapping of the MAX-ACCESS clause

The MAX-ACCESS clause, which must be present, defines whether it makes "protocol sense" to read, write and/or create an instance of the object, or to include its value in a notification. This is the maximal level of access for the object. (This maximal level of access is independent of any administrative authorization policy.)

The value "read-write" indicates that read and write access make "protocol sense", but create does not. The value "read-create" indicates that read, write and create access make "protocol sense". The value "not-accessible" indicates an auxiliary object (see Section 7.7). The value "accessible-for-notify" indicates an object which is accessible only via a notification (e.g., snmpTrapOID [5]).

These values are ordered, from least to greatest: "not-accessible", "accessible-for-notify", "read-only", "read-write", "read-create".

If any columnar object in a conceptual row has "read-create" as its maximal level of access, then no other columnar object of the same conceptual row may have a maximal access of "read-write". (Note that "read-create" is a superset of "read-write".)

7.4. Mapping of the STATUS clause

The STATUS clause, which must be present, indicates whether this definition is current or historic.

The value "current" means that the definition is current and valid. The value "obsolete" means the definition is obsolete and should not be implemented and/or can be removed if previously implemented. While the value "deprecated" also indicates an obsolete definition, it permits new/continued implementation in order to foster interoperability with older/existing implementations.

7.5. Mapping of the DESCRIPTION clause

The DESCRIPTION clause, which must be present, contains a textual definition of that object which provides all semantic definitions necessary for implementation, and should embody any information which would otherwise be communicated in any ASN.1 commentary annotations associated with the object.

7.6. Mapping of the REFERENCE clause

The REFERENCE clause, which need not be present, contains a textual cross-reference to some other document, either another information module which defines a related assignment, or some other document which provides additional information relevant to this definition.

7.7. Mapping of the INDEX clause

The INDEX clause, which must be present if that object corresponds to a conceptual row (unless an AUGMENTS clause is present instead), and must be absent otherwise, defines instance identification information for the columnar objects subordinate to that object.

The instance identification information in an INDEX clause must specify object(s) such that value(s) of those object(s) will unambiguously distinguish a conceptual row. The objects can be columnar objects from the same and/or another conceptual table, but must not be scalar objects. Multiple occurrences of the same object in a single INDEX clause is strongly discouraged.

The syntax of the objects in the INDEX clause indicate how to form the instance-identifier:

(1) integer-valued (i.e., having INTEGER as its underlying primitive type): a single sub-identifier taking the integer value (this works only for non-negative integers);

(2) string-valued, fixed-length strings (or variable-length preceded by the IMPLIED keyword): 'n' sub-identifiers, where 'n' is the length of the string (each octet of the string is encoded in a separate sub-identifier);

(3) string-valued, variable-length strings (not preceded by the IMPLIED keyword): 'n+1' sub-identifiers, where 'n' is the length of the string (the first sub-identifier is 'n' itself, following this, each octet of the string is encoded in a separate sub-identifier);

(4) object identifier-valued (when preceded by the IMPLIED keyword): 'n' sub-identifiers, where 'n' is the number of sub-identifiers in the value (each sub-identifier of the value is copied into a separate sub-identifier);

(5) object identifier-valued (when not preceded by the IMPLIED keyword): 'n+1' sub-identifiers, where 'n' is the number of sub-identifiers in the value (the first sub-identifier is 'n' itself, following this, each sub-identifier in the value is copied);

(6) IpAddress-valued: 4 sub-identifiers, in the familiar a.b.c.d notation.

Note that the IMPLIED keyword can only be present for an object having a variable-length syntax (e.g., variable-length strings or object identifier-valued objects), Further, the IMPLIED keyword can only be associated with the last object in the INDEX clause. Finally, the IMPLIED keyword may not be used on a variable-length string object if that string might have a value of zero-length.

Since a single value of a Counter has (in general) no information content (see section 7.1.6 and 7.1.10), objects defined using the syntax, Counter32 or Counter64, must not be specified in an INDEX

clause. If an object defined using the BITS construct is used in an INDEX clause, it is considered a variable-length string.

Instances identified by use of integer-valued objects should be numbered starting from one (i.e., not from zero). The use of zero as a value for an integer-valued index object should be avoided, except in special cases.

Objects which are both specified in the INDEX clause of a conceptual row and also columnar objects of the same conceptual row are termed auxiliary objects. The MAX-ACCESS clause for auxiliary objects is "not-accessible", except in the following circumstances:

(1) within a MIB module originally written to conform to SMIv1, and later converted to conform to SMIv2; or

(2) a conceptual row must contain at least one columnar object which is not an auxiliary object. In the event that all of a conceptual row's columnar objects are also specified in its INDEX clause, then one of them must be accessible, i.e., have a MAX-ACCESS clause of "read-only". (Note that this situation does not arise for a conceptual row allowing create access, since such a row will have a status column which will not be an auxiliary object.)

Note that objects specified in a conceptual row's INDEX clause need not be columnar objects of that conceptual row. In this situation, the DESCRIPTION clause of the conceptual row must include a textual explanation of how the objects which are included in the INDEX clause but not columnar objects of that conceptual row, are used in uniquely identifying instances of the conceptual row's columnar objects.

7.8. Mapping of the AUGMENTS clause

The AUGMENTS clause, which must not be present unless the object corresponds to a conceptual row, is an alternative to the INDEX clause. Every object corresponding to a conceptual row has either an INDEX clause or an AUGMENTS clause.

If an object corresponding to a conceptual row has an INDEX clause, that row is termed a base conceptual row; alternatively, if the object has an AUGMENTS clause, the row is said to be a conceptual row augmentation, where the AUGMENTS clause names the object corresponding to the base conceptual row which is augmented by this conceptual row augmentation. (Thus, a conceptual row augmentation cannot itself be augmented.) Instances of subordinate columnar objects of a conceptual row augmentation are identified according to the INDEX clause of the base conceptual row corresponding to the object named in the AUGMENTS clause. Further, instances of subordinate columnar objects of a conceptual row augmentation exist according to the same semantics as instances of subordinate columnar objects of the base conceptual row being augmented. As such, note that creation of a base conceptual row implies the correspondent creation of any conceptual row augmentations.

For example, a MIB designer might wish to define additional columns in an "enterprise-specific" MIB which logically extend a conceptual row in a "standard" MIB. The "standard" MIB definition of the conceptual row would include the INDEX clause and the "enterprise-specific" MIB would contain the definition of a conceptual row using the AUGMENTS clause. On the other hand, it would be incorrect to use the AUGMENTS clause for the relationship between RFC 2233's ifTable

and the many media-specific MIBs which extend it for specific media (e.g., the dot3Table in RFC 2358), since not all interfaces are of the same media.

Note that a base conceptual row may be augmented by multiple conceptual row augmentations.

7.8.1. Relation between INDEX and AUGMENTS clauses

When defining instance identification information for a conceptual table:

(1) If there is a one-to-one correspondence between the conceptual rows of this table and an existing table, then the AUGMENTS clause should be used.

(2) Otherwise, if there is a sparse relationship between the conceptual rows of this table and an existing table, then an INDEX clause should be used which is identical to that in the existing table. For example, the relationship between RFC 2233's ifTable and a media-specific MIB which extends the ifTable for a specific media (e.g., the dot3Table in RFC 2358), is a sparse relationship.

(3) Otherwise, if no existing objects have the required syntax and semantics, then auxiliary objects should be defined within the conceptual row for the new table, and those objects should be used within the INDEX clause for the conceptual row.

7.9. Mapping of the DEFVAL clause

The DEFVAL clause, which need not be present, defines an acceptable default value which may be used at the discretion of an agent when an object instance is created. That is, the value is a "hint" to implementors.

During conceptual row creation, if an instance of a columnar object is not present as one of the operands in the correspondent management protocol set operation, then the value of the DEFVAL clause, if present, indicates an acceptable default value that an agent might use (especially for a read-only object).

Note that with this definition of the DEFVAL clause, it is appropriate to use it for any columnar object of a read-create table. It is also permitted to use it for scalar objects dynamically created by an agent, or for columnar objects of a read-write table dynamically created by an agent.

The value of the DEFVAL clause must, of course, correspond to the SYNTAX clause for the object. If the value is an OBJECT IDENTIFIER, then it must be expressed as a single ASN.1 identifier, and not as a collection of sub-identifiers.

Note that if an operand to the management protocol set operation is an instance of a read-only object, then the error 'notWritable' [6] will be returned. As such, the DEFVAL clause can be used to provide an acceptable default value that an agent might use.

By way of example, consider the following possible DEFVAL clauses:

```
ObjectSyntax       DEFVAL clause
----------------   ------------
Integer32          DEFVAL { 1 }
                   -- same for Gauge32, TimeTicks, Unsigned32
INTEGER            DEFVAL { valid } -- enumerated value
OCTET STRING       DEFVAL { 'ffffffffffff'H }
DisplayString      DEFVAL { "SNMP agent" }
IpAddress          DEFVAL { 'c0210415'H } -- 192.33.4.21
OBJECT IDENTIFIER  DEFVAL { sysDescr }
BITS               DEFVAL { { primary, secondary } }
                   -- enumerated values that are set
BITS               DEFVAL { { } }
                   -- no enumerated values are set
```

A binary string used in a DEFVAL clause for an OCTET STRING must be either an integral multiple of eight or zero bits in length; similarly, a hexadecimal string must be an even number of hexadecimal digits. The value of a character string used in a DEFVAL clause must not contain tab characters or line terminator characters.

Object types with SYNTAX of Counter32 and Counter64 may not have DEFVAL clauses, since they do not have defined initial values. However, it is recommended that they be initialized to zero.

7.10. Mapping of the OBJECT-TYPE value

The value of an invocation of the OBJECT-TYPE macro is the name of the object, which is an OBJECT IDENTIFIER, an administratively assigned name.

When an OBJECT IDENTIFIER is assigned to an object:

(1) If the object corresponds to a conceptual table, then only a single assignment, that for a conceptual row, is present immediately beneath that object. The administratively assigned name for the conceptual row object is derived by appending a sub-identifier of

"1" to the administratively assigned name for the conceptual table.

(2) If the object corresponds to a conceptual row, then at least one assignment, one for each column in the conceptual row, is present beneath that object. The administratively assigned name for each column is derived by appending a unique, positive sub-identifier to the administratively assigned name for the conceptual row.

(3) Otherwise, no other OBJECT IDENTIFIERs which are subordinate to the object may be assigned.

Note that the final sub-identifier of any administratively assigned name for an object shall be positive. A zero-valued final sub-identifier is reserved for future use.

7.11. Usage Example

Consider how one might define a conceptual table and its subordinates. (This example uses the RowStatus textual convention defined in [3].)

```
evalSlot OBJECT-TYPE
    SYNTAX      Integer32 (0..2147483647)
    MAX-ACCESS  read-only
    STATUS      current
    DESCRIPTION
            "The index number of the first unassigned entry in the
            evaluation table, or the value of zero indicating that
            all entries are assigned.

            A management station should create new entries in the
            evaluation table using this algorithm:  first, issue a
            management protocol retrieval operation to determine the
            value of evalSlot; and, second, issue a management
            protocol set operation to create an instance of the
            evalStatus object setting its value to createAndGo(4) or
            createAndWait(5).  If this latter operation succeeds,
            then the management station may continue modifying the
            instances corresponding to the newly created conceptual
            row, without fear of collision with other management
            stations."
   ::= { eval 1 }

evalTable OBJECT-TYPE
    SYNTAX      SEQUENCE OF EvalEntry
    MAX-ACCESS  not-accessible
    STATUS      current
    DESCRIPTION
```

```
                "The (conceptual) evaluation table."
        ::= { eval 2 }

    evalEntry OBJECT-TYPE
        SYNTAX      EvalEntry
        MAX-ACCESS  not-accessible
        STATUS      current
        DESCRIPTION
                "An entry (conceptual row) in the evaluation table."
       INDEX   { evalIndex }
       ::= { evalTable 1 }

    EvalEntry ::=
        SEQUENCE {
            evalIndex       Integer32,
            evalString      DisplayString,
            evalValue       Integer32,
            evalStatus      RowStatus
        }

    evalIndex OBJECT-TYPE
        SYNTAX      Integer32 (1..2147483647)
        MAX-ACCESS  not-accessible
        STATUS      current
        DESCRIPTION
                "The auxiliary variable used for identifying instances of
                the columnar objects in the evaluation table."
            ::= { evalEntry 1 }

    evalString OBJECT-TYPE
        SYNTAX      DisplayString
        MAX-ACCESS  read-create
        STATUS      current
        DESCRIPTION
                "The string to evaluate."
            ::= { evalEntry 2 }

    evalValue OBJECT-TYPE
        SYNTAX      Integer32
        MAX-ACCESS  read-only
        STATUS      current
        DESCRIPTION
                "The value when evalString was last evaluated, or zero if
                 no such value is available."
        DEFVAL  { 0 }
            ::= { evalEntry 3 }

    evalStatus OBJECT-TYPE
```

```
        SYNTAX      RowStatus
        MAX-ACCESS  read-create
        STATUS      current
        DESCRIPTION
                "The status column used for creating, modifying, and
                deleting instances of the columnar objects in the
                evaluation table."
    DEFVAL  { active }
        ::= { evalEntry 4 }
```

8. Mapping of the NOTIFICATION-TYPE macro

The NOTIFICATION-TYPE macro is used to define the information contained within an unsolicited transmission of management information (i.e., within either a SNMPv2-Trap-PDU or InformRequest-PDU). It should be noted that the expansion of the NOTIFICATION-TYPE macro is something which conceptually happens during implementation and not during run-time.

8.1. Mapping of the OBJECTS clause

The OBJECTS clause, which need not be present, defines an ordered sequence of MIB object types. One and only one object instance for each occurrence of each object type must be present, and in the specified order, in every instance of the notification. If the same object type occurs multiple times in a notification's ordered sequence, then an object instance is present for each of them. An object type specified in this clause must not have an MAX-ACCESS clause of "not-accessible". The notification's DESCRIPTION clause must specify the information/meaning conveyed by each occurrence of each object type in the sequence. The DESCRIPTION clause must also specify which object instance is present for each object type in the notification.

Note that an agent is allowed, at its own discretion, to append as many additional objects as it considers useful to the end of the notification (i.e., after the objects defined by the OBJECTS clause).

8.2. Mapping of the STATUS clause

The STATUS clause, which must be present, indicates whether this definition is current or historic.

The value "current" means that the definition is current and valid. The value "obsolete" means the definition is obsolete and should not be implemented and/or can be removed if previously implemented. While the value "deprecated" also indicates an obsolete definition, it permits new/continued implementation in order to foster

interoperability with older/existing implementations.

8.3. Mapping of the DESCRIPTION clause

The DESCRIPTION clause, which must be present, contains a textual definition of the notification which provides all semantic definitions necessary for implementation, and should embody any information which would otherwise be communicated in any ASN.1 commentary annotations associated with the notification. In particular, the DESCRIPTION clause should document which instances of the objects mentioned in the OBJECTS clause should be contained within notifications of this type.

8.4. Mapping of the REFERENCE clause

The REFERENCE clause, which need not be present, contains a textual cross-reference to some other document, either another information module which defines a related assignment, or some other document which provides additional information relevant to this definition.

8.5. Mapping of the NOTIFICATION-TYPE value

The value of an invocation of the NOTIFICATION-TYPE macro is the name of the notification, which is an OBJECT IDENTIFIER, an administratively assigned name. In order to achieve compatibility with SNMPv1 traps, both when converting SMIv1 information modules to/from this SMI, and in the procedures employed by multi-lingual systems and proxy forwarding applications, the next to last sub-identifier in the name of any newly-defined notification must have the value zero.

Sections 4.2.6 and 4.2.7 of [6] describe how the NOTIFICATION-TYPE macro is used to generate a SNMPv2-Trap-PDU or InformRequest-PDU, respectively.

8.6. Usage Example

Consider how a configuration change notification might be described:

```
entityMIBTraps      OBJECT IDENTIFIER ::= { entityMIB 2 }
entityMIBTrapPrefix OBJECT IDENTIFIER ::= { entityMIBTraps 0 }

entConfigChange NOTIFICATION-TYPE
    STATUS             current
    DESCRIPTION
            "An entConfigChange trap is sent when the value of
            entLastChangeTime changes. It can be utilized by an NMS to
            trigger logical/physical entity table maintenance polls.
```

```
            An agent must not generate more than one entConfigChange
            'trap-event' in a five second period, where a 'trap-event'
            is the transmission of a single trap PDU to a list of
            trap destinations.  If additional configuration changes
            occur within the five second 'throttling' period, then
            these trap-events should be suppressed by the agent. An
            NMS should periodically check the value of
            entLastChangeTime to detect any missed entConfigChange
            trap-events, e.g. due to throttling or transmission loss."
      ::= { entityMIBTrapPrefix 1 }
```

According to this invocation, the notification authoritatively identified as

```
     { entityMIBTrapPrefix 1 }
```

is used to report a particular type of configuration change.

9. Refined Syntax

Some macros have clauses which allows syntax to be refined, specifically: the SYNTAX clause of the OBJECT-TYPE macro, and the SYNTAX/WRITE-SYNTAX clauses of the MODULE-COMPLIANCE and AGENT-CAPABILITIES macros [2]. However, not all refinements of syntax are appropriate. In particular, the object's primitive or application type must not be changed.

Further, the following restrictions apply:

	Restrictions to Refinement of		
object syntax	range	enumeration	size
INTEGER	(1)	(2)	-
Integer32	(1)	-	-
Unsigned32	(1)	-	-
OCTET STRING	-	-	(3)
OBJECT IDENTIFIER	-	-	-
BITS	-	(2)	-
IpAddress	-	-	-
Counter32	-	-	-
Counter64	-	-	-
Gauge32	(1)	-	-
TimeTicks	-	-	-

where:

(1) the range of permitted values may be refined by raising the lower-bounds, by reducing the upper-bounds, and/or by reducing the alternative value/range choices;

(2) the enumeration of named-values may be refined by removing one or more named-values (note that for BITS, a refinement may cause the enumerations to no longer be contiguous); or,

(3) the size in octets of the value may be refined by raising the lower-bounds, by reducing the upper-bounds, and/or by reducing the alternative size choices.

No other types of refinements can be specified in the SYNTAX clause. However, the DESCRIPTION clause is available to specify additional restrictions which can not be expressed in the SYNTAX clause. Further details on (and examples of) sub-typing are provided in Appendix A.

10. Extending an Information Module

As experience is gained with an information module, it may be desirable to revise that information module. However, changes are not allowed if they have any potential to cause interoperability problems "over the wire" between an implementation using an original specification and an implementation using an updated specification(s).

For any change, the invocation of the MODULE-IDENTITY macro must be updated to include information about the revision: specifically, updating the LAST-UPDATED clause, adding a pair of REVISION and DESCRIPTION clauses (see section 5.5), and making any necessary changes to existing clauses, including the ORGANIZATION and CONTACT-INFO clauses.

Note that any definition contained in an information module is available to be IMPORT-ed by any other information module, and is referenced in an IMPORTS clause via the module name. Thus, a module name should not be changed. Specifically, the module name (e.g., "FIZBIN-MIB" in the example of Section 5.7) should not be changed when revising an information module (except to correct typographical errors), and definitions should not be moved from one information module to another.

Also note that obsolete definitions must not be removed from MIB modules since their descriptors may still be referenced by other information modules, and the OBJECT IDENTIFIERs used to name them must never be re-assigned.

10.1. Object Assignments

If any non-editorial change is made to any clause of a object assignment, then the OBJECT IDENTIFIER value associated with that object assignment must also be changed, along with its associated descriptor.

10.2. Object Definitions

An object definition may be revised in any of the following ways:

(1) A SYNTAX clause containing an enumerated INTEGER may have new enumerations added or existing labels changed. Similarly, named bits may be added or existing labels changed for the BITS construct.

(2) The value of a SYNTAX clause may be replaced by a textual convention, providing the textual convention is defined to use the same primitive ASN.1 type, has the same set of values, and has identical semantics.

(3) A STATUS clause value of "current" may be revised as "deprecated" or "obsolete". Similarly, a STATUS clause value of "deprecated" may be revised as "obsolete". When making such a change, the DESCRIPTION clause should be updated to explain the rationale.

(4) A DEFVAL clause may be added or updated.

(5) A REFERENCE clause may be added or updated.

(6) A UNITS clause may be added.

(7) A conceptual row may be augmented by adding new columnar objects at the end of the row, and making the corresponding update to the SEQUENCE definition.

(8) Clarifications and additional information may be included in the DESCRIPTION clause.

(9) Entirely new objects may be defined, named with previously unassigned OBJECT IDENTIFIER values.

Otherwise, if the semantics of any previously defined object are changed (i.e., if a non-editorial change is made to any clause other than those specifically allowed above), then the OBJECT IDENTIFIER value associated with that object must also be changed.

Note that changing the descriptor associated with an existing object is considered a semantic change, as these strings may be used in an IMPORTS statement.

10.3. Notification Definitions

A notification definition may be revised in any of the following ways:

(1) A REFERENCE clause may be added or updated.

(2) A STATUS clause value of "current" may be revised as "deprecated" or "obsolete". Similarly, a STATUS clause value of "deprecated" may be revised as "obsolete". When making such a change, the DESCRIPTION clause should be updated to explain the rationale.

(3) A DESCRIPTION clause may be clarified.

Otherwise, if the semantics of any previously defined notification are changed (i.e., if a non-editorial change is made to any clause other those specifically allowed above), then the OBJECT IDENTIFIER value associated with that notification must also be changed.

Note that changing the descriptor associated with an existing notification is considered a semantic change, as these strings may be used in an IMPORTS statement.

11. Appendix A: Detailed Sub-typing Rules

11.1. Syntax Rules

The syntax rules for sub-typing are given below. Note that while this syntax is based on ASN.1, it includes some extensions beyond what is allowed in ASN.1, and a number of ASN.1 constructs are not allowed by this syntax.

```
<integerSubType>
    ::= <empty>
      | "(" <range> ["|" <range>]... ")"

<octetStringSubType>
    ::= <empty>
      | "(" "SIZE" "(" <range> ["|" <range>]... ")" ")"

<range>
    ::= <value>
      | <value> ".." <value>

<value>
    ::= "-" <number>
      | <number>
      | <hexString>
      | <binString>

where:
    <empty>     is the empty string
    <number>    is a non-negative integer
    <hexString> is a hexadecimal string (e.g., '0F0F'H)
    <binString> is a binary string (e.g, '1010'B)

    <range> is further restricted as follows:
        - any <value> used in a SIZE clause must be non-negative.
        - when a pair of values is specified, the first value
          must be less than the second value.
        - when multiple ranges are specified, the ranges may
          not overlap but may touch. For example, (1..4 | 4..9)
          is invalid, and (1..4 | 5..9) is valid.
        - the ranges must be a subset of the maximum range of the
          base type.
```

11.2. Examples

Some examples of legal sub-typing:

```
Integer32 (-20..100)
Integer32 (0..100 | 300..500)
Integer32 (300..500 | 0..100)
Integer32 (0 | 2 | 4 | 6 | 8 | 10)
OCTET STRING (SIZE(0..100))
OCTET STRING (SIZE(0..100 | 300..500))
OCTET STRING (SIZE(0 | 2 | 4 | 6 | 8 | 10))
SYNTAX   TimeInterval (0..100)
SYNTAX   DisplayString (SIZE(0..32))
```

(Note the last two examples above are not valid in a TEXTUAL CONVENTION, see [3].)

Some examples of illegal sub-typing:

```
Integer32 (150..100)         -- first greater than second
Integer32 (0..100 | 50..500) -- ranges overlap
Integer32 (0 | 2 | 0 )       -- value duplicated
Integer32 (MIN..-1 | 1..MAX) -- MIN and MAX not allowed
Integer32 (SIZE (0..34))     -- must not use SIZE
OCTET STRING (0..100)        -- must use SIZE
OCTET STRING (SIZE(-10..100)) -- negative SIZE
```

12. Security Considerations

This document defines a language with which to write and read descriptions of management information. The language itself has no security impact on the Internet.

13. Editors' Addresses

Keith McCloghrie
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134-1706
USA
Phone: +1 408 526 5260
EMail: kzm@cisco.com

David Perkins
SNMPinfo
3763 Benton Street
Santa Clara, CA 95051
USA
Phone: +1 408 221-8702
EMail: dperkins@snmpinfo.com

Juergen Schoenwaelder
TU Braunschweig
Bueltenweg 74/75
38106 Braunschweig
Germany
Phone: +49 531 391-3283
EMail: schoenw@ibr.cs.tu-bs.de

14. References

[1] Information processing systems - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1), International Organization for Standardization. International Standard 8824, (December, 1987).

[2] McCloghrie, K., Perkins, D., Schoenwaelder, J., Case, J., Rose, M. and S. Waldbusser, "Conformance Statements for SMIv2", STD 58, RFC 2580, April 1999.

[3] McCloghrie, K., Perkins, D., Schoenwaelder, J., Case, J., Rose, M. and S. Waldbusser, "Textual Conventions for SMIv2", STD 58, RFC 2579, April 1999.

[4] Information processing systems - Open Systems Interconnection - Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1), International Organization for Standardization. International Standard 8825, (December, 1987).

[5] The SNMPv2 Working Group, Case, J., McCloghrie, K., Rose, M. and S. Waldbusser, "Management Information Base for Version 2 of the Simple Network Management Protocol (SNMPv2)", RFC 1907, January 1996.

[6] The SNMPv2 Working Group, Case, J., McCloghrie, K., Rose, M. and S. Waldbusser, "Protocol Operations for Version 2 of the Simple Network Management Protocol (SNMPv2)", RFC 1905, January 1996.

15. Full Copyright Statement

Network Working Group
Request for Comments: 2579
STD: 58
Obsoletes: 1903
Category: Standards Track

Editors of this version:
K. McCloghrie
Cisco Systems
D. Perkins
SNMPinfo
J. Schoenwaelder
TU Braunschweig
Authors of previous version:
J. Case
SNMP Research
K. McCloghrie
Cisco Systems
M. Rose
First Virtual Holdings
S. Waldbusser
International Network Services
April 1999

Textual Conventions for SMIv2

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction

Management information is viewed as a collection of managed objects, residing in a virtual information store, termed the Management Information Base (MIB). Collections of related objects are defined in MIB modules. These modules are written using an adapted subset of OSI's Abstract Syntax Notation One, ASN.1 (1988) [1], termed the Structure of Management Information (SMI) [2].

When designing a MIB module, it is often useful to define new types similar to those defined in the SMI. In comparison to a type defined in the SMI, each of these new types has a different name, a similar syntax, but a more precise semantics. These newly defined types are termed textual conventions, and are used for the convenience of humans reading the MIB module. It is the purpose of this document to define the initial set of textual conventions available to all MIB modules.

Objects defined using a textual convention are always encoded by means of the rules that define their primitive type. However, textual conventions often have special semantics associated with them. As such, an ASN.1 macro, TEXTUAL-CONVENTION, is used to concisely convey the syntax and semantics of a textual convention.

1.1. A Note on Terminology

For the purpose of exposition, the original Structure of Management Information, as described in RFCs 1155 (STD 16), 1212 (STD 16), and RFC 1215, is termed the SMI version 1 (SMIv1). The current version of the Structure of Management Information is termed SMI version 2 (SMIv2).

2. Definitions

```
SNMPv2-TC DEFINITIONS ::= BEGIN

IMPORTS
    TimeTicks         FROM SNMPv2-SMI;


-- definition of textual conventions

TEXTUAL-CONVENTION MACRO ::=
```

```
BEGIN
    TYPE NOTATION ::=
                  DisplayPart
                  "STATUS" Status
                  "DESCRIPTION" Text
                  ReferPart
                  "SYNTAX" Syntax

    VALUE NOTATION ::=
                   value(VALUE Syntax)      -- adapted ASN.1

    DisplayPart ::=
                  "DISPLAY-HINT" Text
                | empty

    Status ::=
                  "current"
                | "deprecated"
                | "obsolete"

    ReferPart ::=
                  "REFERENCE" Text
                | empty

    -- a character string as defined in [2]
    Text ::= value(IA5String)

    Syntax ::=   -- Must be one of the following:
                      -- a base type (or its refinement), or
                      -- a BITS pseudo-type
                  type
                | "BITS" "{" NamedBits "}"

    NamedBits ::= NamedBit
                | NamedBits "," NamedBit

    NamedBit ::=  identifier "(" number ")" -- number is nonnegative

END



DisplayString ::= TEXTUAL-CONVENTION
    DISPLAY-HINT "255a"
    STATUS       current
    DESCRIPTION
            "Represents textual information taken from the NVT ASCII
```

```
            character set, as defined in pages 4, 10-11 of RFC 854.

            To summarize RFC 854, the NVT ASCII repertoire specifies:

              - the use of character codes 0-127 (decimal)

              - the graphics characters (32-126) are interpreted as
                US ASCII

              - NUL, LF, CR, BEL, BS, HT, VT and FF have the special
                meanings specified in RFC 854

              - the other 25 codes have no standard interpretation

              - the sequence 'CR LF' means newline

              - the sequence 'CR NUL' means carriage-return

              - an 'LF' not preceded by a 'CR' means moving to the
                same column on the next line.

              - the sequence 'CR x' for any x other than LF or NUL is
                illegal.  (Note that this also means that a string may
                end with either 'CR LF' or 'CR NUL', but not with CR.)

            Any object defined using this syntax may not exceed 255
            characters in length."
        SYNTAX       OCTET STRING (SIZE (0..255))

PhysAddress ::= TEXTUAL-CONVENTION
        DISPLAY-HINT "1x:"
        STATUS       current
        DESCRIPTION
                "Represents media- or physical-level addresses."
        SYNTAX       OCTET STRING


MacAddress ::= TEXTUAL-CONVENTION
        DISPLAY-HINT "1x:"
        STATUS       current
        DESCRIPTION
                "Represents an 802 MAC address represented in the
                'canonical' order defined by IEEE 802.1a, i.e., as if it
                were transmitted least significant bit first, even though
                802.5 (in contrast to other 802.x protocols) requires MAC
                addresses to be transmitted most significant bit first."
        SYNTAX       OCTET STRING (SIZE (6))
```

```
TruthValue ::= TEXTUAL-CONVENTION
    STATUS       current
    DESCRIPTION
            "Represents a boolean value."
    SYNTAX       INTEGER { true(1), false(2) }

TestAndIncr ::= TEXTUAL-CONVENTION
    STATUS       current
    DESCRIPTION
            "Represents integer-valued information used for atomic
            operations.  When the management protocol is used to specify
            that an object instance having this syntax is to be
            modified, the new value supplied via the management protocol
            must precisely match the value presently held by the
            instance.  If not, the management protocol set operation
            fails with an error of 'inconsistentValue'.  Otherwise, if
            the current value is the maximum value of 2^31-1 (2147483647
            decimal), then the value held by the instance is wrapped to
            zero; otherwise, the value held by the instance is
            incremented by one.  (Note that regardless of whether the
            management protocol set operation succeeds, the variable-
            binding in the request and response PDUs are identical.)

            The value of the ACCESS clause for objects having this
            syntax is either 'read-write' or 'read-create'.  When an
            instance of a columnar object having this syntax is created,
            any value may be supplied via the management protocol.

            When the network management portion of the system is re-
            initialized, the value of every object instance having this
            syntax must either be incremented from its value prior to
            the re-initialization, or (if the value prior to the re-
            initialization is unknown) be set to a pseudo-randomly
            generated value."
    SYNTAX       INTEGER (0..2147483647)

AutonomousType ::= TEXTUAL-CONVENTION
    STATUS       current
    DESCRIPTION
            "Represents an independently extensible type identification
            value.  It may, for example, indicate a particular sub-tree
            with further MIB definitions, or define a particular type of
            protocol or hardware."
    SYNTAX       OBJECT IDENTIFIER


InstancePointer ::= TEXTUAL-CONVENTION
    STATUS       obsolete
```

```
    DESCRIPTION
            "A pointer to either a specific instance of a MIB object or
            a conceptual row of a MIB table in the managed device.  In
            the latter case, by convention, it is the name of the
            particular instance of the first accessible columnar object
            in the conceptual row.

            The two uses of this textual convention are replaced by
            VariablePointer and RowPointer, respectively."
    SYNTAX       OBJECT IDENTIFIER


VariablePointer ::= TEXTUAL-CONVENTION
    STATUS       current
    DESCRIPTION
            "A pointer to a specific object instance.  For example,
            sysContact.0 or ifInOctets.3."
    SYNTAX       OBJECT IDENTIFIER


RowPointer ::= TEXTUAL-CONVENTION
    STATUS       current
    DESCRIPTION
            "Represents a pointer to a conceptual row.  The value is the
            name of the instance of the first accessible columnar object
            in the conceptual row.

            For example, ifIndex.3 would point to the 3rd row in the
            ifTable (note that if ifIndex were not-accessible, then
            ifDescr.3 would be used instead)."
    SYNTAX       OBJECT IDENTIFIER

RowStatus ::= TEXTUAL-CONVENTION
    STATUS       current
    DESCRIPTION
            "The RowStatus textual convention is used to manage the
            creation and deletion of conceptual rows, and is used as the
            value of the SYNTAX clause for the status column of a
            conceptual row (as described in Section 7.7.1 of [2].)
```

The status column has six defined values:

- 'active', which indicates that the conceptual row is available for use by the managed device;

- 'notInService', which indicates that the conceptual row exists in the agent, but is unavailable for use by the managed device (see NOTE below); 'notInService' has no implication regarding the internal consistency of the row, availability of resources, or consistency with the current state of the managed device;

- 'notReady', which indicates that the conceptual row exists in the agent, but is missing information necessary in order to be available for use by the managed device (i.e., one or more required columns in the conceptual row have not been instanciated);

- 'createAndGo', which is supplied by a management station wishing to create a new instance of a conceptual row and to have its status automatically set to active, making it available for use by the managed device;

- 'createAndWait', which is supplied by a management station wishing to create a new instance of a conceptual row (but not make it available for use by the managed device); and,

- 'destroy', which is supplied by a management station wishing to delete all of the instances associated with an existing conceptual row.

Whereas five of the six values (all except 'notReady') may be specified in a management protocol set operation, only three values will be returned in response to a management protocol retrieval operation: 'notReady', 'notInService' or 'active'. That is, when queried, an existing conceptual row has only three states: it is either available for use by the managed device (the status column has value 'active'); it is not available for use by the managed device, though the agent has sufficient information to attempt to make it so (the status column has value 'notInService'); or, it is not available for use by the managed device, and an attempt to make it so would fail because the agent has insufficient information (the state column has value 'notReady').

NOTE WELL

> This textual convention may be used for a MIB table, irrespective of whether the values of that table's conceptual rows are able to be modified while it is active, or whether its conceptual rows must be taken out of service in order to be modified. That is, it is the responsibility of the DESCRIPTION clause of the status column to specify whether the status column must not be 'active' in order for the value of some other column of the same conceptual row to be modified. If such a specification is made, affected columns may be changed by an SNMP set PDU if the RowStatus would not be equal to 'active' either immediately before or after processing the PDU. In other words, if the PDU also contained a varbind that would change the RowStatus value, the column in question may be changed if the RowStatus was not equal to 'active' as the PDU was received, or if the varbind sets the status to a value other than 'active'.

Also note that whenever any elements of a row exist, the RowStatus column must also exist.

To summarize the effect of having a conceptual row with a status column having a SYNTAX clause value of RowStatus, consider the following state diagram:

	STATE			
	A	B	C	D
ACTION	status column does not exist	status col. is notReady	status column is notInService	status column is active
set status column to createAndGo	noError or inconsistent-Value ->D	inconsist-entValue	inconsistent-Value	inconsistent-Value
set status column to createAndWait	noError or wrongValue see 1	inconsist-entValue	inconsistent-Value	inconsistent-Value
set status column to active	inconsistent-Value	inconsist-entValue or see 2 ->D	noError see 8 ->D	noError ->D
set status column to notInService	inconsistent-Value	inconsist-entValue or see 3 ->C	noError ->C	noError ->C or see 6
set status column to destroy	noError ->A	noError ->A	noError ->A	noError ->A or see 7
set any other column to some value	see 4	noError see 1	noError ->C	see 5 ->D

(1) goto B or C, depending on information available to the agent.

(2) if other variable bindings included in the same PDU,

provide values for all columns which are missing but required, and all columns have acceptable values, then return noError and goto D.

(3) if other variable bindings included in the same PDU, provide legal values for all columns which are missing but required, then return noError and goto C.

(4) at the discretion of the agent, the return value may be either:

inconsistentName: because the agent does not choose to create such an instance when the corresponding RowStatus instance does not exist, or

inconsistentValue: if the supplied value is inconsistent with the state of some other MIB object's value, or

noError: because the agent chooses to create the instance.

If noError is returned, then the instance of the status column must also be created, and the new state is B or C, depending on the information available to the agent. If inconsistentName or inconsistentValue is returned, the row remains in state A.

(5) depending on the MIB definition for the column/table, either noError or inconsistentValue may be returned.

(6) the return value can indicate one of the following errors:

wrongValue: because the agent does not support notInService (e.g., an agent which does not support createAndWait), or

inconsistentValue: because the agent is unable to take the row out of service at this time, perhaps because it is in use and cannot be de-activated.

(7) the return value can indicate the following error:

inconsistentValue: because the agent is unable to remove the row at this time, perhaps because it is in use and cannot be de-activated.

(8) the transition to D can fail, e.g., if the values of the conceptual row are inconsistent, then the error code would be inconsistentValue.

NOTE: Other processing of (this and other varbinds of) the set request may result in a response other than noError being returned, e.g., wrongValue, noCreation, etc.

Conceptual Row Creation

There are four potential interactions when creating a conceptual row: selecting an instance-identifier which is not in use; creating the conceptual row; initializing any objects for which the agent does not supply a default; and, making the conceptual row available for use by the managed device.

Interaction 1: Selecting an Instance-Identifier

The algorithm used to select an instance-identifier varies for each conceptual row. In some cases, the instance-identifier is semantically significant, e.g., the destination address of a route, and a management station selects the instance-identifier according to the semantics.

In other cases, the instance-identifier is used solely to distinguish conceptual rows, and a management station without specific knowledge of the conceptual row might examine the instances present in order to determine an unused instance-identifier. (This approach may be used, but it is often highly sub-optimal; however, it is also a questionable practice for a naive management station to attempt conceptual row creation.)

Alternately, the MIB module which defines the conceptual row might provide one or more objects which provide assistance in determining an unused instance-identifier. For example, if the conceptual row is indexed by an integer-value, then an object having an integer-valued SYNTAX clause might be defined for such a purpose, allowing a management station to issue a management protocol retrieval operation. In order to avoid unnecessary collisions between competing management stations, ‘adjacent’ retrievals of this object should be different.

Finally, the management station could select a pseudo-random number to use as the index. In the event that this index

was already in use and an inconsistentValue was returned in response to the management protocol set operation, the management station should simply select a new pseudo-random number and retry the operation.

A MIB designer should choose between the two latter algorithms based on the size of the table (and therefore the efficiency of each algorithm). For tables in which a large number of entries are expected, it is recommended that a MIB object be defined that returns an acceptable index for creation. For tables with small numbers of entries, it is recommended that the latter pseudo-random index mechanism be used.

Interaction 2: Creating the Conceptual Row

Once an unused instance-identifier has been selected, the management station determines if it wishes to create and activate the conceptual row in one transaction or in a negotiated set of interactions.

Interaction 2a: Creating and Activating the Conceptual Row

The management station must first determine the column requirements, i.e., it must determine those columns for which it must or must not provide values. Depending on the complexity of the table and the management station's knowledge of the agent's capabilities, this determination can be made locally by the management station. Alternately, the management station issues a management protocol get operation to examine all columns in the conceptual row that it wishes to create. In response, for each column, there are three possible outcomes:

- a value is returned, indicating that some other management station has already created this conceptual row. We return to interaction 1.

- the exception 'noSuchInstance' is returned, indicating that the agent implements the object-type associated with this column, and that this column in at least one conceptual row would be accessible in the MIB view used by the retrieval were it to exist. For those columns to which the agent provides read-create access, the 'noSuchInstance' exception tells the management station that it should supply a value for this column when the conceptual row is to be created.

- the exception 'noSuchObject' is returned, indicating that the agent does not implement the object-type associated with this column or that there is no conceptual row for which this column would be accessible in the MIB view used by the retrieval. As such, the management station can not issue any management protocol set operations to create an instance of this column.

Once the column requirements have been determined, a management protocol set operation is accordingly issued. This operation also sets the new instance of the status column to 'createAndGo'.

When the agent processes the set operation, it verifies that it has sufficient information to make the conceptual row available for use by the managed device. The information available to the agent is provided by two sources: the management protocol set operation which creates the conceptual row, and, implementation-specific defaults supplied by the agent (note that an agent must provide implementation-specific defaults for at least those objects which it implements as read-only). If there is sufficient information available, then the conceptual row is created, a 'noError' response is returned, the status column is set to 'active', and no further interactions are necessary (i.e., interactions 3 and 4 are skipped). If there is insufficient information, then the conceptual row is not created, and the set operation fails with an error of 'inconsistentValue'. On this error, the management station can issue a management protocol retrieval operation to determine if this was because it failed to specify a value for a required column, or, because the selected instance of the status column already existed. In the latter case, we return to interaction 1. In the former case, the management station can re-issue the set operation with the additional information, or begin interaction 2 again using 'createAndWait' in order to negotiate creation of the conceptual row.

NOTE WELL

> Regardless of the method used to determine the column requirements, it is possible that the management station might deem a column necessary when, in fact, the agent will not allow that particular columnar instance to be created or written. In this case, the management protocol set operation will fail with an error such as 'noCreation' or 'notWritable'. In this case, the management station decides whether it needs to be able to set a value for that particular columnar instance. If not, the management station re-issues the management protocol set operation, but without setting a value for that particular columnar instance; otherwise, the management station aborts the row creation algorithm.

Interaction 2b: Negotiating the Creation of the Conceptual Row

The management station issues a management protocol set operation which sets the desired instance of the status column to 'createAndWait'. If the agent is unwilling to process a request of this sort, the set operation fails with an error of 'wrongValue'. (As a consequence, such an agent must be prepared to accept a single management protocol set operation, i.e., interaction 2a above, containing all of the columns indicated by its column requirements.) Otherwise, the conceptual row is created, a 'noError' response is returned, and the status column is immediately set to either 'notInService' or 'notReady', depending on whether it has sufficient information to (attempt to) make the conceptual row available for use by the managed device. If there is sufficient information available, then the status column is set to 'notInService'; otherwise, if there is insufficient information, then the status column is set to 'notReady'. Regardless, we proceed to interaction 3.

Interaction 3: Initializing non-defaulted Objects

The management station must now determine the column requirements. It issues a management protocol get operation to examine all columns in the created conceptual row. In the response, for each column, there are three possible outcomes:

- a value is returned, indicating that the agent implements the object-type associated with this column and had sufficient information to provide a value. For those columns to which the agent provides read-create access (and for which the agent allows their values to be changed after their creation), a value return tells the management station that it may issue additional management protocol set operations, if it desires, in order to change the value associated with this column.

- the exception 'noSuchInstance' is returned, indicating that the agent implements the object-type associated with this column, and that this column in at least one conceptual row would be accessible in the MIB view used by the retrieval were it to exist. However, the agent does not have sufficient information to provide a value, and until a value is provided, the conceptual row may not be made available for use by the managed device. For those columns to which the agent provides read-create access, the 'noSuchInstance' exception tells the management station that it must issue additional management protocol set operations, in order to provide a value associated with this column.

- the exception 'noSuchObject' is returned, indicating that the agent does not implement the object-type associated with this column or that there is no conceptual row for which this column would be accessible in the MIB view used by the retrieval. As such, the management station can not issue any management protocol set operations to create an instance of this column.

If the value associated with the status column is 'notReady', then the management station must first deal with all 'noSuchInstance' columns, if any. Having done so, the value of the status column becomes 'notInService', and we proceed to interaction 4.

Interaction 4: Making the Conceptual Row Available

Once the management station is satisfied with the values associated with the columns of the conceptual row, it issues a management protocol set operation to set the status column to 'active'. If the agent has sufficient information to make the conceptual row available for use by the managed device, the management protocol set operation succeeds (a 'noError' response is returned). Otherwise, the management protocol set operation fails with an error of 'inconsistentValue'.

NOTE WELL

> A conceptual row having a status column with value 'notInService' or 'notReady' is unavailable to the managed device. As such, it is possible for the managed device to create its own instances during the time between the management protocol set operation which sets the status column to 'createAndWait' and the management protocol set operation which sets the status column to 'active'. In this case, when the management protocol set operation is issued to set the status column to 'active', the values held in the agent supersede those used by the managed device.

If the management station is prevented from setting the status column to 'active' (e.g., due to management station or network failure) the conceptual row will be left in the 'notInService' or 'notReady' state, consuming resources indefinitely. The agent must detect conceptual rows that have been in either state for an abnormally long period of time and remove them. It is the responsibility of the DESCRIPTION clause of the status column to indicate what an abnormally long period of time would be. This period of time should be long enough to allow for human response time (including 'think time') between the creation of the conceptual row and the setting of the status to 'active'. In the absence of such information in the DESCRIPTION clause, it is suggested that this period be approximately 5 minutes in length. This removal action applies not only to newly-created rows, but also to previously active rows which are set to, and left in, the notInService state for a prolonged period exceeding that which is considered normal for such a conceptual row.

Conceptual Row Suspension

When a conceptual row is 'active', the management station may issue a management protocol set operation which sets the instance of the status column to 'notInService'. If the agent is unwilling to do so, the set operation fails with an error of 'wrongValue' or 'inconsistentValue'. Otherwise, the conceptual row is taken out of service, and a 'noError' response is returned. It is the responsibility of the DESCRIPTION clause of the status column to indicate under what circumstances the status column should be taken out of service (e.g., in order for the value of some other column of the same conceptual row to be modified).

Conceptual Row Deletion

For deletion of conceptual rows, a management protocol set operation is issued which sets the instance of the status column to 'destroy'. This request may be made regardless of the current value of the status column (e.g., it is possible to delete conceptual rows which are either 'notReady', 'notInService' or 'active'.) If the operation succeeds, then all instances associated with the conceptual row are immediately removed."

```
    SYNTAX      INTEGER {
                    -- the following two values are states:
                    -- these values may be read or written
                    active(1),
                    notInService(2),

                    -- the following value is a state:
                    -- this value may be read, but not written
                    notReady(3),

                    -- the following three values are
                    -- actions: these values may be written,
                    --   but are never read
                    createAndGo(4),
                    createAndWait(5),
                    destroy(6)
                }

TimeStamp ::= TEXTUAL-CONVENTION
    STATUS      current
    DESCRIPTION
            "The value of the sysUpTime object at which a specific
            occurrence happened.  The specific occurrence must be
```

```
            defined in the description of any object defined using this
            type.

            If sysUpTime is reset to zero as a result of a re-
            initialization of the network management (sub)system, then
            the values of all TimeStamp objects are also reset.
            However, after approximately 497 days without a re-
            initialization, the sysUpTime object will reach 2^^32-1 and
            then increment around to zero; in this case, existing values
            of TimeStamp objects do not change.  This can lead to
            ambiguities in the value of TimeStamp objects."
    SYNTAX      TimeTicks


TimeInterval ::= TEXTUAL-CONVENTION
    STATUS      current
    DESCRIPTION
            "A period of time, measured in units of 0.01 seconds."
    SYNTAX      INTEGER (0..2147483647)

DateAndTime ::= TEXTUAL-CONVENTION
    DISPLAY-HINT "2d-1d-1d,1d:1d:1d.1d,1a1d:1d"
    STATUS      current
    DESCRIPTION
            "A date-time specification.

            field  octets  contents                  range
            -----  ------  --------                  -----
              1      1-2   year*                     0..65536
              2       3    month                     1..12
              3       4    day                       1..31
              4       5    hour                      0..23
              5       6    minutes                   0..59
              6       7    seconds                   0..60
                           (use 60 for leap-second)
              7       8    deci-seconds              0..9
              8       9    direction from UTC        '+' / '-'
              9      10    hours from UTC*           0..13
             10      11    minutes from UTC          0..59

            * Notes:
            - the value of year is in network-byte order
            - daylight saving time in New Zealand is +13

            For example, Tuesday May 26, 1992 at 1:30:15 PM EDT would be
            displayed as:

                             1992-5-26,13:30:15.0,-4:0
```

```
            Note that if only local time is known, then timezone
            information (fields 8-10) is not present."
    SYNTAX      OCTET STRING (SIZE (8 | 11))


StorageType ::= TEXTUAL-CONVENTION
    STATUS      current
    DESCRIPTION
            "Describes the memory realization of a conceptual row.  A
            row which is volatile(2) is lost upon reboot.  A row which
            is either nonVolatile(3), permanent(4) or readOnly(5), is
            backed up by stable storage.  A row which is permanent(4)
            can be changed but not deleted.  A row which is readOnly(5)
            cannot be changed nor deleted.

            If the value of an object with this syntax is either
            permanent(4) or readOnly(5), it cannot be written.
            Conversely, if the value is either other(1), volatile(2) or
            nonVolatile(3), it cannot be modified to be permanent(4) or
            readOnly(5).  (All illegal modifications result in a
            'wrongValue' error.)

            Every usage of this textual convention is required to
            specify the columnar objects which a permanent(4) row must
            at a minimum allow to be writable."
    SYNTAX      INTEGER {
                    other(1),       -- eh?
                    volatile(2),    -- e.g., in RAM
                    nonVolatile(3), -- e.g., in NVRAM
                    permanent(4),   -- e.g., partially in ROM
                    readOnly(5)     -- e.g., completely in ROM
                }
```

```
TDomain ::= TEXTUAL-CONVENTION
    STATUS       current
    DESCRIPTION
          "Denotes a kind of transport service.

          Some possible values, such as snmpUDPDomain, are defined in
          the SNMPv2-TM MIB module.  Other possible values are defined
          in other MIB modules."
    REFERENCE    "The SNMPv2-TM MIB module is defined in RFC 1906."
    SYNTAX       OBJECT IDENTIFIER


TAddress ::= TEXTUAL-CONVENTION
    STATUS       current
    DESCRIPTION
          "Denotes a transport service address.

          A TAddress value is always interpreted within the context of a
          TDomain value.  Thus, each definition of a TDomain value must
          be accompanied by a definition of a textual convention for use
          with that TDomain.  Some possible textual conventions, such as
          SnmpUDPAddress for snmpUDPDomain, are defined in the SNMPv2-TM
          MIB module.  Other possible textual conventions are defined in
          other MIB modules."
    REFERENCE    "The SNMPv2-TM MIB module is defined in RFC 1906."
    SYNTAX       OCTET STRING (SIZE (1..255))


END
```

3. Mapping of the TEXTUAL-CONVENTION macro

The TEXTUAL-CONVENTION macro is used to convey the syntax and semantics associated with a textual convention. It should be noted that the expansion of the TEXTUAL-CONVENTION macro is something which conceptually happens during implementation and not during run-time.

The name of a textual convention must consist of one or more letters or digits, with the initial character being an upper case letter. The name must not conflict with any of the reserved words listed in section 3.7 of [2], should not consist of all upper case letters, and shall not exceed 64 characters in length. (However, names longer than 32 characters are not recommended.) The hyphen is not allowed in the name of a textual convention (except for use in information modules converted from SMIv1 which allowed hyphens in ASN.1 type assignments). Further, all names used for the textual conventions defined in all "standard" information modules shall be unique.

3.1. Mapping of the DISPLAY-HINT clause

The DISPLAY-HINT clause, which need not be present, gives a hint as to how the value of an instance of an object with the syntax defined using this textual convention might be displayed. The DISPLAY-HINT clause must not be present if the Textual Convention is defined with a syntax of: OBJECT IDENTIFIER, IpAddress, Counter32, Counter64, or any enumerated syntax (BITS or INTEGER). The determination of whether it makes sense for other syntax types is dependent on the specific definition of the Textual Convention.

When the syntax has an underlying primitive type of INTEGER, the hint consists of an integer-format specification, containing two parts. The first part is a single character suggesting a display format, either: ‘x’ for hexadecimal, or ‘d’ for decimal, or ‘o’ for octal, or ‘b’ for binary. For all types, when rendering the value, leading zeros are omitted, and for negative values, a minus sign is rendered immediately before the digits. The second part is always omitted for ‘x’, ‘o’ and ‘b’, and need not be present for ‘d’. If present, the second part starts with a hyphen and is followed by a decimal number, which defines the implied decimal point when rendering the value. For example:

```
     Hundredths ::= TEXTUAL-CONVENTION
         DISPLAY-HINT "d-2"
         ...
         SYNTAX     INTEGER (0..10000)
```

suggests that a Hundredths value of 1234 be rendered as "12.34"

When the syntax has an underlying primitive type of OCTET STRING, the hint consists of one or more octet-format specifications. Each specification consists of five parts, with each part using and removing zero or more of the next octets from the value and producing the next zero or more characters to be displayed. The octets within the value are processed in order of significance, most significant first.

The five parts of a octet-format specification are:

(1) the (optional) repeat indicator; if present, this part is a ‘*’, and indicates that the current octet of the value is to be used as the repeat count. The repeat count is an unsigned integer (which may be zero) which specifies how many times the remainder of this octet-format specification should be successively applied. If the repeat indicator is not present, the repeat count is one.

(2) the octet length: one or more decimal digits specifying the number of octets of the value to be used and formatted by this octet-specification. Note that the octet length can be zero. If less than this number of octets remain in the value, then the lesser number of octets are used.

(3) the display format, either: 'x' for hexadecimal, 'd' for decimal, 'o' for octal, 'a' for ascii, or 't' for UTF-8. If the octet length part is greater than one, and the display format part refers to a numeric format, then network-byte ordering (big-endian encoding) is used interpreting the octets in the value. The octets processed by the 't' display format do not necessarily form an integral number of UTF-8 characters. Trailing octets which do not form a valid UTF-8 encoded character are discarded.

(4) the (optional) display separator character; if present, this part is a single character which is produced for display after each application of this octet-specification; however, this character is not produced for display if it would be immediately followed by the display of the repeat terminator character for this octet-specification. This character can be any character other than a decimal digit and a '*'.

(5) the (optional) repeat terminator character, which can be present only if the display separator character is present and this octet-specification begins with a repeat indicator; if present, this part is a single character which is produced after all the zero or more repeated applications (as given by the repeat count) of this octet-specification. This character can be any character other than a decimal digit and a '*'.

Output of a display separator character or a repeat terminator character is suppressed if it would occur as the last character of the display.

If the octets of the value are exhausted before all the octet-format specification have been used, then the excess specifications are ignored. If additional octets remain in the value after interpreting all the octet-format specifications, then the last octet-format specification is re-interpreted to process the additional octets, until no octets remain in the value.

3.2. Mapping of the STATUS clause

The STATUS clause, which must be present, indicates whether this definition is current or historic.

The value "current" means that the definition is current and valid.

The value "obsolete" means the definition is obsolete and should not be implemented and/or can be removed if previously implemented. While the value "deprecated" also indicates an obsolete definition, it permits new/continued implementation in order to foster interoperability with older/existing implementations.

3.3. Mapping of the DESCRIPTION clause

The DESCRIPTION clause, which must be present, contains a textual definition of the textual convention, which provides all semantic definitions necessary for implementation, and should embody any information which would otherwise be communicated in any ASN.1 commentary annotations associated with the object.

3.4. Mapping of the REFERENCE clause

The REFERENCE clause, which need not be present, contains a textual cross-reference to some other document, either another information module which defines a related assignment, or some other document which provides additional information relevant to this definition.

3.5. Mapping of the SYNTAX clause

The SYNTAX clause, which must be present, defines abstract data structure corresponding to the textual convention. The data structure must be one of the alternatives defined in the ObjectSyntax CHOICE or the BITS construct (see section 7.1 in [2]). Note that this means that the SYNTAX clause of a Textual Convention can not refer to a previously defined Textual Convention.

An extended subset of the full capabilities of ASN.1 (1988) sub-typing is allowed, as appropriate to the underlying ASN.1 type. Any such restriction on size, range or enumerations specified in this clause represents the maximal level of support which makes "protocol sense". Restrictions on sub-typing are specified in detail in Section 9 and Appendix A of [2].

4. Sub-typing of Textual Conventions

The SYNTAX clause of a TEXTUAL CONVENTION macro may be sub-typed in the same way as the SYNTAX clause of an OBJECT-TYPE macro (see section 11 of [2]).

5. Revising a Textual Convention Definition

It may be desirable to revise the definition of a textual convention after experience is gained with it. However, changes are not allowed if they have any potential to cause interoperability problems "over

the wire" between an implementation using an original specification and an implementation using an updated specification(s). Such changes can only be accommodated by defining a new textual convention (i.e., a new name).

The following revisions are allowed:

(1) A SYNTAX clause containing an enumerated INTEGER may have new enumerations added or existing labels changed. Similarly, named bits may be added or existing labels changed for the BITS construct.

(2) A STATUS clause value of "current" may be revised as "deprecated" or "obsolete". Similarly, a STATUS clause value of "deprecated" may be revised as "obsolete". When making such a change, the DESCRIPTION clause should be updated to explain the rationale.

(3) A REFERENCE clause may be added or updated.

(4) A DISPLAY-HINTS clause may be added or updated.

(5) Clarifications and additional information may be included in the DESCRIPTION clause.

(6) Any editorial change.

Note that with the introduction of the TEXTUAL-CONVENTION macro, there is no longer any need to define types in the following manner:

```
DisplayString ::= OCTET STRING (SIZE (0..255))
```

When revising an information module containing a definition such as this, that definition should be replaced by a TEXTUAL-CONVENTION macro.

6. Security Considerations

This document defines the means to define new data types for the language used to write and read descriptions of management information. These data types have no security impact on the Internet.

7. Editors' Addresses

Keith McCloghrie
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134-1706
USA
Phone: +1 408 526 5260
EMail: kzm@cisco.com

David Perkins
SNMPinfo
3763 Benton Street
Santa Clara, CA 95051
USA
Phone: +1 408 221-8702
EMail: dperkins@snmpinfo.com

Juergen Schoenwaelder
TU Braunschweig
Bueltenweg 74/75
38106 Braunschweig
Germany
Phone: +49 531 391-3283
EMail: schoenw@ibr.cs.tu-bs.de

8. References

[1] Information processing systems - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1), International Organization for Standardization. International Standard 8824, (December, 1987).

[2] McCloghrie, K., Perkins, D., Schoenwaelder, J., Case, J., Rose, M. and S. Waldbusser, "Structure of Management Information Version 2 (SMIv2)", STD 58, RFC 2578, April 1999.

[3] The SNMPv2 Working Group, Case, J., McCloghrie, K., Rose, M. and Waldbusser, S., "Transport Mappings for Version 2 of the" Simple Network Management Protocol (SNMPv2)", RFC 1906, January 1996.

9. Full Copyright Statement

Network Working Group
Request for Comments: 2580
STD: 58
Obsoletes: 1904
Category: Standards Track

Editors of this version:
K. McCloghrie
Cisco Systems
D. Perkins
SNMPinfo
J. Schoenwaelder
TU Braunschweig
Authors of previous version:
J. Case
SNMP Research
K. McCloghrie
Cisco Systems
M. Rose
First Virtual Holdings
S. Waldbusser
International Network Services
April 1999

Conformance Statements for SMIv2

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction

Management information is viewed as a collection of managed objects, residing in a virtual information store, termed the Management Information Base (MIB). Collections of related objects are defined in MIB modules. These modules are written using an adapted subset of OSI's Abstract Syntax Notation One, ASN.1 (1988) [1], termed the Structure of Management Information (SMI) [2].

It may be useful to define the acceptable lower-bounds of implementation, along with the actual level of implementation achieved. It is the purpose of this document to define the notation used for these purposes.

1.1. A Note on Terminology

For the purpose of exposition, the original Structure of Management Information, as described in RFCs 1156 (STD 16), 1212 (STD 16), and RFC 1215, is termed the SMI version 1 (SMIv1). The current version of the Structure of Management Information is termed SMI version 2 (SMIv2).

2. Definitions

```
SNMPv2-CONF DEFINITIONS ::= BEGIN

IMPORTS ObjectName, NotificationName, ObjectSyntax
                                            FROM SNMPv2-SMI;

-- definitions for conformance groups

OBJECT-GROUP MACRO ::=
BEGIN
    TYPE NOTATION ::=
                  ObjectsPart
                  "STATUS" Status
                  "DESCRIPTION" Text
                  ReferPart

    VALUE NOTATION ::=
                  value(VALUE OBJECT IDENTIFIER)

    ObjectsPart ::=
                  "OBJECTS" "{" Objects "}"
    Objects ::=
                  Object
                | Objects "," Object
    Object ::=
```

```
                value(ObjectName)

    Status ::=
                "current"
              | "deprecated"
              | "obsolete"

    ReferPart ::=
                "REFERENCE" Text
              | empty

    -- a character string as defined in [2]
    Text ::= value(IA5String)
END

-- more definitions for conformance groups

NOTIFICATION-GROUP MACRO ::=
BEGIN
    TYPE NOTATION ::=
                NotificationsPart
                "STATUS" Status
                "DESCRIPTION" Text
                ReferPart

    VALUE NOTATION ::=
                value(VALUE OBJECT IDENTIFIER)

    NotificationsPart ::=
                "NOTIFICATIONS" "{" Notifications "}"
    Notifications ::=
                Notification
              | Notifications "," Notification
    Notification ::=
                value(NotificationName)

    Status ::=
                "current"
              | "deprecated"
              | "obsolete"

    ReferPart ::=
                "REFERENCE" Text
              | empty

    -- a character string as defined in [2]
    Text ::= value(IA5String)
END
```

```
-- definitions for compliance statements

MODULE-COMPLIANCE MACRO ::=
BEGIN
    TYPE NOTATION ::=
                  "STATUS" Status
                  "DESCRIPTION" Text
                  ReferPart
                  ModulePart

    VALUE NOTATION ::=
                  value(VALUE OBJECT IDENTIFIER)

    Status ::=
                  "current"
                | "deprecated"
                | "obsolete"

    ReferPart ::=
                  "REFERENCE" Text
                | empty

    ModulePart ::=
                  Modules
    Modules ::=
                  Module
                | Modules Module
    Module ::=
                  -- name of module --
                  "MODULE" ModuleName
                  MandatoryPart
                  CompliancePart

    ModuleName ::=
                  -- identifier must start with uppercase letter
                  identifier ModuleIdentifier
                  -- must not be empty unless contained
                  -- in MIB Module
                | empty
    ModuleIdentifier ::=
                  value(OBJECT IDENTIFIER)
                | empty

    MandatoryPart ::=
                  "MANDATORY-GROUPS" "{" Groups "}"
                | empty

    Groups ::=
```

```
                Group
              | Groups "," Group
    Group ::=
                value(OBJECT IDENTIFIER)

    CompliancePart ::=
                Compliances
              | empty

    Compliances ::=
                Compliance
              | Compliances Compliance
    Compliance ::=
                ComplianceGroup
              | Object

    ComplianceGroup ::=
                "GROUP" value(OBJECT IDENTIFIER)
                "DESCRIPTION" Text

    Object ::=
                "OBJECT" value(ObjectName)
                SyntaxPart
                WriteSyntaxPart
                AccessPart
                "DESCRIPTION" Text

    -- must be a refinement for object's SYNTAX clause
    SyntaxPart ::= "SYNTAX" Syntax
              | empty

    -- must be a refinement for object's SYNTAX clause
    WriteSyntaxPart ::= "WRITE-SYNTAX" Syntax
              | empty

    Syntax ::=    -- Must be one of the following:
                       -- a base type (or its refinement),
                       -- a textual convention (or its refinement), or
                       -- a BITS pseudo-type
                  type
              | "BITS" "{" NamedBits "}"

    NamedBits ::= NamedBit
              | NamedBits "," NamedBit

    NamedBit ::= identifier "(" number ")" -- number is nonnegative

    AccessPart ::=
```

```
                  "MIN-ACCESS" Access
                | empty
    Access ::=
                  "not-accessible"
                | "accessible-for-notify"
                | "read-only"
                | "read-write"
                | "read-create"

    -- a character string as defined in [2]
    Text ::= value(IA5String)
END

-- definitions for capabilities statements

AGENT-CAPABILITIES MACRO ::=
BEGIN
    TYPE NOTATION ::=
                  "PRODUCT-RELEASE" Text
                  "STATUS" Status
                  "DESCRIPTION" Text
                  ReferPart
                  ModulePart

    VALUE NOTATION ::=
                  value(VALUE OBJECT IDENTIFIER)

    Status ::=
                  "current"
                | "obsolete"

    ReferPart ::=
                  "REFERENCE" Text
                | empty

    ModulePart ::=
                  Modules
                | empty
    Modules ::=
                  Module
                | Modules Module
    Module ::=
                  -- name of module --
                  "SUPPORTS" ModuleName
                  "INCLUDES" "{" Groups "}"
                  VariationPart

    ModuleName ::=
```

```
                -- identifier must start with uppercase letter
                identifier ModuleIdentifier
    ModuleIdentifier ::=
                value(OBJECT IDENTIFIER)
              | empty

    Groups ::=
                Group
              | Groups "," Group
    Group ::=
                value(OBJECT IDENTIFIER)

    VariationPart ::=
                Variations
              | empty
    Variations ::=
                Variation
              | Variations Variation

    Variation ::=
                ObjectVariation
              | NotificationVariation

    NotificationVariation ::=
                "VARIATION" value(NotificationName)
                AccessPart
                "DESCRIPTION" Text

    ObjectVariation ::=
                "VARIATION" value(ObjectName)
                SyntaxPart
                WriteSyntaxPart
                AccessPart
                CreationPart
                DefValPart
                "DESCRIPTION" Text

    -- must be a refinement for object's SYNTAX clause
    SyntaxPart ::= "SYNTAX" Syntax
              | empty

    WriteSyntaxPart ::= "WRITE-SYNTAX" Syntax
              | empty

    Syntax ::=    -- Must be one of the following:
                       -- a base type (or its refinement),
                       -- a textual convention (or its refinement), or
                       -- a BITS pseudo-type
```

```
                type
              | "BITS" "{" NamedBits "}"

    NamedBits ::= NamedBit
              | NamedBits "," NamedBit

    NamedBit ::= identifier "(" number ")" -- number is nonnegative

    AccessPart ::=
                "ACCESS" Access
              | empty

    Access ::=
                "not-implemented"
              -- only "not-implemented" for notifications
              | "accessible-for-notify"
              | "read-only"
              | "read-write"
              | "read-create"
              -- following is for backward-compatibility only
              | "write-only"

    CreationPart ::=
                "CREATION-REQUIRES" "{" Cells "}"
              | empty
    Cells ::=
                Cell
              | Cells "," Cell
    Cell ::=
                value(ObjectName)

    DefValPart ::= "DEFVAL" "{" Defvalue "}"
              | empty

    Defvalue ::=  -- must be valid for the object's syntax
                -- in this macro's SYNTAX clause, if present,
                -- or if not, in object's OBJECT-TYPE macro
                value(ObjectSyntax)
              | "{" BitsValue "}"

    BitsValue ::= BitNames
              | empty

    BitNames ::=  BitName
              | BitNames "," BitName

    BitName ::= identifier
```

```
    -- a character string as defined in [2]
    Text ::= value(IA5String)
END

END
```

3. Mapping of the OBJECT-GROUP macro

For conformance purposes, it is useful to define a collection of related managed objects. The OBJECT-GROUP macro is used to define each such collection of related objects. It should be noted that the expansion of the OBJECT-GROUP macro is something which conceptually happens during implementation and not during run-time.

To "implement" an object, an agent must return a reasonably accurate value for management protocol retrieval operations; similarly, if the object is writable, then in response to a management protocol set operation, an agent must accordingly be able to reasonably influence the underlying managed entity. If an agent can not implement an object, the management protocol provides for it to return an exception or error, e.g, noSuchObject [4]. Under no circumstances shall an agent return a value for objects which it does not implement -- it must always return the appropriate exception or error, as described in the protocol specification [4].

Note that the OBJECT-GROUP macro itself provides no conformance information. Rather, conformance information is specified through the inclusion of defined groups in a MODULE-COMPLIANCE macro.

3.1. Mapping of the OBJECTS clause

The OBJECTS clause, which must be present, is used to specify each object contained in the conformance group. Each of the specified objects must be defined in the same information module as the OBJECT-GROUP macro appears, and must have a MAX-ACCESS clause value of "accessible-for-notify", "read-only", "read-write", or "read-create".

It is required that every object defined in an information module with a MAX-ACCESS clause other than "not-accessible" be contained in at least one object group. This avoids the common error of adding a new object to an information module and forgetting to add the new object to a group.

3.2. Mapping of the STATUS clause

The STATUS clause, which must be present, indicates whether this definition is current or historic.

The value "current" means that the definition is current and valid. The value "obsolete" means the definition is obsolete and the group should no longer be used for defining conformance. While the value "deprecated" also indicates an obsolete definition, it permits new/continued use of conformance definitions using this group.

3.3. Mapping of the DESCRIPTION clause

The DESCRIPTION clause, which must be present, contains a textual definition of that group, along with a description of any relations to other groups. Note that generic compliance requirements should not be stated in this clause. However, implementation relationships between this group and other groups may be defined in this clause.

3.4. Mapping of the REFERENCE clause

The REFERENCE clause, which need not be present, contains a textual cross-reference to some other document, either another information module which defines a related assignment, or some other document which provides additional information relevant to this definition.

3.5. Mapping of the OBJECT-GROUP value

The value of an invocation of the OBJECT-GROUP macro is the name of the group, which is an OBJECT IDENTIFIER, an administratively assigned name.

3.6. Usage Example

The SNMP Group [3] is described:

```
snmpGroup OBJECT-GROUP
    OBJECTS { snmpInPkts,
              snmpInBadVersions,
              snmpInASNParseErrs,
              snmpBadOperations,
              snmpSilentDrops,
              snmpProxyDrops,
              snmpEnableAuthenTraps }
    STATUS  current
    DESCRIPTION
            "A collection of objects providing basic instrumentation
            and control of an agent."
   ::= { snmpMIBGroups 8 }
```

According to this invocation, the conformance group named

```
     { snmpMIBGroups 8 }
```

contains 7 objects.

4. Mapping of the NOTIFICATION-GROUP macro

For conformance purposes, it is useful to define a collection of notifications. The NOTIFICATION-GROUP macro serves this purpose. It should be noted that the expansion of the NOTIFICATION-GROUP macro is something which conceptually happens during implementation and not during run-time.

4.1. Mapping of the NOTIFICATIONS clause

The NOTIFICATIONS clause, which must be present, is used to specify each notification contained in the conformance group. Each of the specified notifications must be defined in the same information module as the NOTIFICATION-GROUP macro appears.

It is required that every notification defined in an information module be contained in at least one notification group.

4.2. Mapping of the STATUS clause

The STATUS clause, which must be present, indicates whether this definition is current or historic.

The value "current" means that the definition is current and valid. The value "obsolete" means the definition is obsolete and this group should no longer be used for defining conformance. While the value "deprecated" also indicates an obsolete definition, it permits new/continued use of conformance definitions using this group.

4.3. Mapping of the DESCRIPTION clause

The DESCRIPTION clause, which must be present, contains a textual definition of the group, along with a description of any relations to other groups. Note that generic compliance requirements should not be stated in this clause. However, implementation relationships between this group and other groups may be defined in this clause.

4.4. Mapping of the REFERENCE clause

The REFERENCE clause, which need not be present, contains a textual cross-reference to some other document, either another information module which defines a related assignment, or some other document which provides additional information relevant to this definition.

4.5. Mapping of the NOTIFICATION-GROUP value

The value of an invocation of the NOTIFICATION-GROUP macro is the name of the group, which is an OBJECT IDENTIFIER, an administratively assigned name.

4.6. Usage Example

The SNMP Basic Notifications Group [3] is described:

```
snmpBasicNotificationsGroup NOTIFICATION-GROUP
    NOTIFICATIONS { coldStart, authenticationFailure }
    STATUS        current
    DESCRIPTION
            "The two notifications which an agent is required to
            implement."
   ::= { snmpMIBGroups 7 }
```

According to this invocation, the conformance group named

```
{ snmpMIBGroups 7 }
```

contains 2 notifications.

5. Mapping of the MODULE-COMPLIANCE macro

The MODULE-COMPLIANCE macro is used to convey a minimum set of requirements with respect to implementation of one or more MIB modules. It should be noted that the expansion of the MODULE-COMPLIANCE macro is something which conceptually happens during implementation and not during run-time.

A requirement on all "standard" MIB modules is that a corresponding MODULE-COMPLIANCE specification is also defined, either in the same information module or in a companion information module.

5.1. Mapping of the STATUS clause

The STATUS clause, which must be present, indicates whether this definition is current or historic.

The value "current" means that the definition is current and valid. The value "obsolete" means the definition is obsolete, and this MODULE-COMPLIANCE specification no longer specifies a valid definition of conformance. While the value "deprecated" also indicates an obsolete definition, it permits new/continued use of the MODULE-COMPLIANCE specification.

5.2. Mapping of the DESCRIPTION clause

The DESCRIPTION clause, which must be present, contains a textual definition of this compliance statement and should embody any information which would otherwise be communicated in any ASN.1 commentary annotations associated with the statement.

5.3. Mapping of the REFERENCE clause

The REFERENCE clause, which need not be present, contains a textual cross-reference to some other document, either another information module which defines a related assignment, or some other document which provides additional information relevant to this definition.

5.4. Mapping of the MODULE clause

The MODULE clause, which must be present, is repeatedly used to name each MIB module for which compliance requirements are being specified. Each MIB module is named by its module name, and optionally, by its associated OBJECT IDENTIFIER as well. The module name can be omitted when the MODULE-COMPLIANCE invocation occurs inside a MIB module, to refer to the encompassing MIB module.

5.4.1. Mapping of the MANDATORY-GROUPS clause

The MANDATORY-GROUPS clause, which need not be present, names the one or more object or notification groups within the correspondent MIB module which are unconditionally mandatory for implementation. If an agent claims compliance to the MIB module, then it must implement each and every object and notification within each conformance group listed. That is, if an agent returns a noSuchObject exception in response to a management protocol get operation [4] for any object within any mandatory conformance group for every possible MIB view, or if the agent cannot generate each notification listed in any conformance group under the appropriate circumstances, then that agent is not a conformant implementation of the MIB module.

5.4.2. Mapping of the GROUP clause

The GROUP clause, which need not be present, is repeatedly used to name each object and notification group which is conditionally mandatory for compliance to the MIB module. The GROUP clause can also be used to name unconditionally optional groups. A group named in a GROUP clause must be absent from the correspondent MANDATORY-GROUPS clause.

Conditionally mandatory groups include those which are mandatory only if a particular protocol is implemented, or only if another group is implemented. A GROUP clause's DESCRIPTION specifies the conditions under which the group is conditionally mandatory.

A group which is named in neither a MANDATORY-GROUPS clause nor a GROUP clause, is unconditionally optional for compliance to the MIB module.

5.4.3. Mapping of the OBJECT clause

The OBJECT clause, which need not be present, is repeatedly used to specify each MIB object for which compliance has a refined requirement with respect to the MIB module definition. The MIB object must be present in one of the conformance groups named in the correspondent MANDATORY-GROUPS clause or GROUP clauses.

By definition, each object specified in an OBJECT clause follows a MODULE clause which names the information module in which that object is defined. Therefore, the use of an IMPORTS statement, to specify from where such objects are imported, is redundant and is not required in an information module.

5.4.3.1. Mapping of the SYNTAX clause

The SYNTAX clause, which need not be present, is used to provide a refined SYNTAX for the object named in the correspondent OBJECT clause. Note that if this clause and a WRITE-SYNTAX clause are both present, then this clause only applies when instances of the object named in the correspondent OBJECT clause are read.

Consult Section 9 of [2] for more information on refined syntax.

5.4.3.2. Mapping of the WRITE-SYNTAX clause

The WRITE-SYNTAX clause, which need not be present, is used to provide a refined SYNTAX for the object named in the correspondent OBJECT clause when instances of that object are written.

Consult Section 9 of [2] for more information on refined syntax.

5.4.3.3. Mapping of the MIN-ACCESS clause

The MIN-ACCESS clause, which need not be present, is used to define the minimal level of access for the object named in the correspondent OBJECT clause. If this clause is absent, the minimal level of access is the same as the maximal level specified in the correspondent invocation of the OBJECT-TYPE macro. If present, this clause must not specify a greater level of access than is specified in the correspondent invocation of the OBJECT-TYPE macro.

The level of access for certain types of objects is fixed according to their syntax definition. These types include: conceptual tables and rows, auxiliary objects, and objects with the syntax of Counter32, Counter64 (and possibly, certain types of textual conventions). A MIN-ACCESS clause should not be present for such

objects.

An implementation is compliant if the level of access it provides is greater or equal to the minimal level in the MODULE-COMPLIANCE macro and less or equal to the maximal level in the OBJECT-TYPE macro.

5.4.4. Mapping of the DESCRIPTION clause

The DESCRIPTION clause must be present for each use of the GROUP or OBJECT clause. For an OBJECT clause, it contains a textual description of the refined compliance requirement. For a GROUP clause, it contains a textual description of the conditions under which the group is conditionally mandatory or unconditionally optional.

5.5. Mapping of the MODULE-COMPLIANCE value

The value of an invocation of the MODULE-COMPLIANCE macro is an OBJECT IDENTIFIER. As such, this value may be authoritatively used when referring to the compliance statement embodied by that invocation of the macro.

5.6. Usage Example

The compliance statement contained in the (hypothetical) XYZv2-MIB might be:

```
xyzMIBCompliance MODULE-COMPLIANCE
    STATUS  current
    DESCRIPTION
            "The compliance statement for XYZv2 entities which
            implement the XYZv2 MIB."
   MODULE  -- compliance to the containing MIB module
       MANDATORY-GROUPS { xyzSystemGroup,
                          xyzStatsGroup, xyzTrapGroup,
                          xyzSetGroup,
                          xyzBasicNotificationsGroup }

       GROUP   xyzV1Group
       DESCRIPTION
           "The xyzV1 group is mandatory only for those
            XYZv2 entities which also implement XYZv1."
 ::= { xyzMIBCompliances 1 }
```

According to this invocation, to claim alignment with the compliance statement named

```
{ xyzMIBCompliances 1 }
```

a system must implement the XYZv2-MIB's xyzSystemGroup, xyzStatsGroup, xyzTrapGroup, and xyzSetGroup object conformance groups, as well as the xyzBasicNotificationsGroup notifications group. Furthermore, if the XYZv2 entity also implements XYZv1, then it must also support the XYZv1Group group, if compliance is to be claimed.

6. Mapping of the AGENT-CAPABILITIES macro

The AGENT-CAPABILITIES macro is used to convey a set of capabilities present in an agent. It should be noted that the expansion of the AGENT-CAPABILITIES macro is something which conceptually happens during implementation and not during run-time.

When a MIB module is written, it is divided into units of conformance termed groups. If an agent claims to implement a group, then it must implement each and every object, or each and every notification, within that group. Of course, for whatever reason, an agent might implement only a subset of the groups within a MIB module. In addition, the definition of some MIB objects/notifications leave some aspects of the definition to the discretion of an implementor.

Practical experience has demonstrated a need for concisely describing the capabilities of an agent with respect to one or more MIB modules. The AGENT-CAPABILITIES macro allows an agent implementor to describe the precise level of support which an agent claims in regards to a MIB group, and to bind that description to the value of an instance of sysORID [3]. In particular, some objects may have restricted or augmented syntax or access-levels.

If the AGENT-CAPABILITIES invocation is given to a management-station implementor, then that implementor can build management applications which optimize themselves when communicating with a particular agent. For example, the management-station can maintain a database of these invocations. When a management-station interacts with an agent, it retrieves from the agent the values of all instances of sysORID [3]. Based on this, it consults the database to locate each entry matching one of the retrieved values of sysORID. Using the located entries, the management application can now optimize its behavior accordingly.

Note that the AGENT-CAPABILITIES macro specifies refinements or variations with respect to OBJECT-TYPE and NOTIFICATION-TYPE macros in MIB modules, NOT with respect to MODULE-COMPLIANCE macros in compliance statements.

6.1. Mapping of the PRODUCT-RELEASE clause

The PRODUCT-RELEASE clause, which must be present, contains a textual description of the product release which includes this set of capabilities.

6.2. Mapping of the STATUS clause

The STATUS clause, which must be present, indicates whether this

definition is current or historic.

The value "current" means that the definition is current and valid. The value "obsolete" means the definition is obsolete and this capabilities statement is no longer in use.

6.3. Mapping of the DESCRIPTION clause

The DESCRIPTION clause, which must be present, contains a textual description of this set of capabilities.

6.4. Mapping of the REFERENCE clause

The REFERENCE clause, which need not be present, contains a textual cross-reference to some other document, either another information module which defines a related assignment, or some other document which provides additional information relevant to this definition.

6.5. Mapping of the SUPPORTS clause

The SUPPORTS clause, which need not be present, is repeatedly used to name each MIB module for which the agent claims a complete or partial implementation. Each MIB module is named by its module name, and optionally, by its associated OBJECT IDENTIFIER (as registered by the MODULE-IDENTITY macro, see [2]) as well.

6.5.1. Mapping of the INCLUDES clause

The INCLUDES clause, which must follow each and every use of the SUPPORTS clause, is used to name each MIB group associated with the SUPPORTS clause, which the agent claims to implement.

6.5.2. Mapping of the VARIATION clause

The VARIATION clause, which need not be present, is repeatedly used to name each object or notification which the agent implements in some variant or refined fashion with respect to the correspondent invocation of the OBJECT-TYPE or NOTIFICATION-TYPE macro.

Note that the variation concept is meant for generic implementation restrictions, e.g., if the variation for an object depends on the values of other objects, then this should be noted in the appropriate DESCRIPTION clause.

By definition, each object specified in a VARIATION clause follows a SUPPORTS clause which names the information module in which that object is defined. Therefore, the use of an IMPORTS statement, to specify from where such objects are imported, is redundant and is not

required in an information module.

6.5.2.1. Mapping of the SYNTAX clause

The SYNTAX clause, which need not be present, is used to provide a refined SYNTAX for the object named in the correspondent VARIATION clause. Note that if this clause and a WRITE-SYNTAX clause are both present, then this clause only applies when instances of the object named in the correspondent VARIATION clause are read.

Consult Section 9 of [2] for more information on refined syntax.

Note that for enumerated INTEGERs and for the BITS construct, the changes allowed when updating a MIB module include the addition of enumerations and/or changing the labels of existing enumerations (see Section 10.2 of [2]). This type of change can cause problems for an AGENT-CAPABILITIES macro written against the old revision of a MIB module. One way to avoid such problems is to explicitly list all objects having an enumerated syntax in a VARIATION clause, even when all enumerations are currently supported.

6.5.2.2. Mapping of the WRITE-SYNTAX clause

The WRITE-SYNTAX clause, which need not be present, is used to provide a refined SYNTAX for the object named in the correspondent VARIATION clause when instances of that object are written.

Consult Section 9 of [2] for more information on refined syntax.

6.5.2.3. Mapping of the ACCESS clause

The ACCESS clause, which need not be present, is used to indicate the agent provides less than the maximal level of access to the object or notification named in the correspondent VARIATION clause.

The only value applicable to notifications is "not-implemented".

The value "not-implemented" indicates the agent does not implement the object or notification, and in the ordering of possible values is equivalent to "not-accessible".

The value "write-only" is provided solely for backward compatibility, and shall not be used for newly-defined object types. In the ordering of possible values, "write-only" is less than "not-accessible".

6.5.2.4. Mapping of the CREATION-REQUIRES clause

The CREATION-REQUIRES clause, which need not be present, is used to name the columnar objects of a conceptual row to which values must be explicitly assigned, by a management protocol set operation, before the agent will allow the instance of the status column of that row to be set to 'active'. (Consult the definition of RowStatus [5].)

If the conceptual row does not have a status column (i.e., the objects corresponding to the conceptual table were defined using the mechanisms in [6,7]), then the CREATION-REQUIRES clause, which need not be present, is used to name the columnar objects of a conceptual row to which values must be explicitly assigned, by a management protocol set operation, before the agent will create new instances of objects in that row.

This clause must not be present unless the object named in the correspondent VARIATION clause is a conceptual row, i.e., has a syntax which resolves to a SEQUENCE containing columnar objects. The objects named in the value of this clause usually will refer to columnar objects in that row. However, objects unrelated to the conceptual row may also be specified.

All objects which are named in the CREATION-REQUIRES clause for a conceptual row, and which are columnar objects of that row, must have an access level of "read-create".

6.5.2.5. Mapping of the DEFVAL clause

The DEFVAL clause, which need not be present, is used to provide a alternate DEFVAL value for the object named in the correspondent VARIATION clause. The semantics of this value are identical to those of the OBJECT-TYPE macro's DEFVAL clause.

6.5.2.6. Mapping of the DESCRIPTION clause

The DESCRIPTION clause, which must be present for each use of the VARIATION clause, contains a textual description of the variant or refined implementation of the object or notification.

6.6. Mapping of the AGENT-CAPABILITIES value

The value of an invocation of the AGENT-CAPABILITIES macro is an OBJECT IDENTIFIER, which names the value of sysORID [3] for which this capabilities statement is valid.

6.7. Usage Example

Consider how a capabilities statement for an agent might be described:

```
exampleAgent AGENT-CAPABILITIES
    PRODUCT-RELEASE      "ACME Agent release 1.1 for 4BSD."
    STATUS               current
    DESCRIPTION          "ACME agent for 4BSD."

    SUPPORTS             SNMPv2-MIB
        INCLUDES         { systemGroup, snmpGroup, snmpSetGroup,
                           snmpBasicNotificationsGroup }

        VARIATION        coldStart
            DESCRIPTION  "A coldStart trap is generated on all
                         reboots."

    SUPPORTS             IF-MIB
        INCLUDES         { ifGeneralGroup, ifPacketGroup }

        VARIATION        ifAdminStatus
            SYNTAX       INTEGER { up(1), down(2) }
            DESCRIPTION  "Unable to set test mode on 4BSD."

        VARIATION        ifOperStatus
            SYNTAX       INTEGER { up(1), down(2) }
            DESCRIPTION  "Information limited on 4BSD."

    SUPPORTS             IP-MIB
        INCLUDES         { ipGroup, icmpGroup }

        VARIATION        ipDefaultTTL
            SYNTAX       INTEGER (255..255)
            DESCRIPTION  "Hard-wired on 4BSD."

        VARIATION        ipInAddrErrors
            ACCESS       not-implemented
            DESCRIPTION  "Information not available on 4BSD."

        VARIATION        ipNetToMediaEntry
            CREATION-REQUIRES { ipNetToMediaPhysAddress }
            DESCRIPTION  "Address mappings on 4BSD require
                         both protocol and media addresses."

    SUPPORTS             TCP-MIB
        INCLUDES         { tcpGroup }
        VARIATION        tcpConnState
```

```
            ACCESS       read-only
            DESCRIPTION  "Unable to set this on 4BSD."

        SUPPORTS             UDP-MIB
            INCLUDES         { udpGroup }

        SUPPORTS             EVAL-MIB
            INCLUDES         { functionsGroup, expressionsGroup }
            VARIATION        exprEntry
                CREATION-REQUIRES { evalString, evalStatus }
                DESCRIPTION  "Conceptual row creation is supported."

        ::= { acmeAgents 1 }
```

According to this invocation, an agent with a sysORID value of

```
     { acmeAgents 1 }
```

supports objects defined in six MIB modules.

From SNMPv2-MIB, five conformance groups are supported.

From IF-MIB, the ifGeneralGroup and ifPacketGroup groups are supported. However, the objects ifAdminStatus and ifOperStatus have a restricted syntax.

From IP-MIB, all objects in the ipGroup and icmpGroup are supported except ipInAddrErrors, while ipDefaultTTL has a restricted range, and when creating a new instance in the ipNetToMediaTable, the set-request must create an instance of ipNetToMediaPhysAddress.

From TCP-MIB, the tcpGroup is supported except that tcpConnState is available only for reading.

From UDP-MIB, the udpGroup is fully supported.

From the EVAL-MIB, all the objects contained in the functionsGroup and expressionsGroup conformance groups are supported, without variation. In addition, creation of new instances in the expr table is supported, and requires both of the objects: evalString and evalStatus, to be assigned a value.

7. Extending an Information Module

As experience is gained with a published information module, it may be desirable to revise that information module.

Section 10 of [2] defines the rules for extending an information module. The remainder of this section defines how conformance groups, compliance statements, and capabilities statements may be extended.

7.1. Conformance Groups

It may be desirable to revise the definition of a conformance group (an OBJECT-GROUP or a NOTIFICATION-GROUP) after experience is gained with it. However, conformance groups can be referenced by compliance and/or capabilities definitions. Therefore, a change to a conformance group is not allowed if it has the potential to cause a reference to the group's original definition to be different from a reference to the updated definition. Such changes can only be accommodated by defining a new conformance group with a new descriptor and a new OBJECT IDENTIFIER value.

The following revisions are allowed:

(1) A STATUS clause value of "current" may be revised as "deprecated" or "obsolete". Similarly, a STATUS clause value of "deprecated" may be revised as "obsolete". When making such a change, the DESCRIPTION clause should be updated to explain the rationale.

(2) A REFERENCE clause may be added or updated.

(3) Clarifications and additional information may be included in the DESCRIPTION clause.

(4) Any editorial change.

It is not necessary to change the STATUS value of a conformance group when the status of a member of the group is changed.

7.2. Compliance Definitions

It may be desirable to revise the definition of a compliance definition (MODULE-COMPLIANCE) after experience is gained with it. However, changes are not allowed if they cause the requirements specified by the original definition to be different from the requirements of the updated definition. Such changes can only be accommodated by defining a new compliance definition with a new

descriptor and a new OBJECT IDENTIFIER value.

The following revisions are allowed:

(1) A STATUS clause value of "current" may be revised as "deprecated" or "obsolete". Similarly, a STATUS clause value of "deprecated" may be revised as "obsolete". When making such a change, the DESCRIPTION clause should be updated to explain the rationale.

(2) A REFERENCE clause may be added or updated.

(3) Clarifications and additional information may be included in the DESCRIPTION clause(s).

(4) Any editorial change.

It is not necessary to change the STATUS value of a compliance definition due to a change in the STATUS value of a definition it references.

7.3. Capabilities Definitions

It may be desirable to revise the definition of a capabilities definition (AGENT-CAPABILITIES) after experience is gained with it. However, changes are not allowed if they cause the capabilities specified by the original specification to be different from the capabilities of the updated specification. Such changes can only be accommodated by defining a new capabilities definition with a new descriptor and a new OBJECT IDENTIFIER value.

The following revisions are allowed:

(1) A STATUS clause value of "current" may be revised as "obsolete". When making such a change, the DESCRIPTION clause should be updated to explain the rationale.

(2) A REFERENCE clause may be added or updated.

(3) Clarifications and additional information may be included in the DESCRIPTION clause(s).

(4) Any editorial change.

It is not necessary to change the STATUS value of a capabilities definition due to a change in the STATUS value of a definition it references.

8. Security Considerations

This document defines the means to define conformance requirements for implementing on documents describing management information. This method of defining conformance requirements has no security impact on the Internet.

9. Editors' Addresses

Keith McCloghrie
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134-1706
USA
Phone: +1 408 526 5260
EMail: kzm@cisco.com

David Perkins
SNMPinfo
3763 Benton Street
Santa Clara, CA 95051
USA
Phone: +1 408 221-8702
Email: dperkins@snmpinfo.com

Juergen Schoenwaelder
TU Braunschweig
Bueltenweg 74/75
38106 Braunschweig
Germany
Phone: +49 531 391-3283
EMail: schoenw@ibr.cs.tu-bs.de

10. References

[1] Information processing systems - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1), International Organization for Standardization. International Standard 8824, (December, 1987).

[2] McCloghrie, K., Perkins, D., Schoenwaelder, J., Case, J., Rose, M. and S. Waldbusser, "Structure of Management Information Version 2 (SMIv2)", STD 58, RFC 2578, April 1999.

[3] The SNMPv2 Working Group, Case, J., McCloghrie, K., Rose, M. and S. Waldbusser, "Management Information Base for Version 2 of the Simple Network Management Protocol (SNMPv2)", RFC 1907, January 1996.

[4] The SNMPv2 Working Group, Case, J., McCloghrie, K., Rose, M. and S. Waldbusser, "Protocol Operations for Version 2 of the Simple Network Management Protocol (SNMPv2)", RFC 1905, January 1996.

[5] McCloghrie, K., Perkins, D., Schoenwaelder, J., Case, J., Rose, M. and S. Waldbusser, "Textual Conventions for SMIv2", STD 58, RFC 2579, April 1999.

[6] Rose, M. and K. McCloghrie, "Structure and Identification of Management Information for TCP/IP-based internets", STD 16, RFC 1155, May 1990.

[7] Rose, M. and K. McCloghrie, "Concise MIB Definitions", STD 16, RFC 1212, March 1991.

11. Full Copyright Statement