Network Working Group
Request for Comments: 1902
Obsoletes: 1442
Category: Standards Track

SNMPv2 Working Group
J. Case
SNMP Research, Inc.
K. McCloghrie
Cisco Systems, Inc.
M. Rose
Dover Beach Consulting, Inc.
S. Waldbusser
International Network Services

January 1996

# Structure of Management Information for Version 2 of the Simple Network Management Protocol (SNMPv2)

## Status of this Memo



## 1. Introduction

A management system contains: several (potentially many) nodes, each with a processing entity, termed an agent, which has access to management instrumentation; at least one management station; and, a management protocol, used to convey management information between the agents and management stations. Operations of the protocol are carried out under an administrative framework which defines authentication, authorization, access control, and privacy policies.

Management stations execute management applications which monitor and control managed elements. Managed elements are devices such as hosts, routers, terminal servers, etc., which are monitored and controlled via access to their management information.

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) [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 Internet-standard Network Management Framework, as described in RFCs 1155 (STD 16), 1157 (STD 15), and 1212 (STD 16), is termed the SNMP version 1 framework (SNMPv1). The current framework is termed the SNMP version 2 framework (SNMPv2).

## 2. Definitions

```
SNMPv2-SMI DEFINITIONS ::= BEGIN


-- the path to the root

org             OBJECT IDENTIFIER ::= { iso 3 }
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 }


-- definitions for information modules

MODULE-IDENTITY MACRO ::=
BEGIN
    TYPE NOTATION ::=
                  "LAST-UPDATED" value(Update UTCTime)
                  "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 UTCTime)
                  "DESCRIPTION" Text

    -- uses the NVT ASCII character set
    Text ::= """" string """"
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

    Text ::= """" string """"
END


-- names of objects

ObjectName ::=
    OBJECT IDENTIFIER

NotificationName ::=
    OBJECT IDENTIFIER

-- syntax of objects

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 ::=
    [UNIVERSAL 2]
        IMPLICIT 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 ::=
```

```
                type(ObjectSyntax)
              | "BITS" "{" Kibbles "}"
    Kibbles ::=
                Kibble
              | Kibbles "," Kibble
    Kibble ::=
                identifier "(" nonNegativeNumber ")"

    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(Indexobject ObjectName)
    Entry ::=
                  -- use the INDEX value of the
                  -- correspondent OBJECT-TYPE invocation
                value(Entryobject ObjectName)

    DefValPart ::=
```

```
                  "DEFVAL" "{" value(Defval Syntax) "}"
                | empty

    -- uses the NVT ASCII character set
    Text ::= """" string """"
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(Name ObjectName)

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

    ReferPart ::=
                "REFERENCE" Text
              | empty

    -- uses the NVT ASCII character set
    Text ::= """" string """"
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 a subset of ASN.1 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 SNMPv2 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 SNMPv2, and instances of ASN.1 macros defined in this document and in other documents [2, 3] of the SNMPv2 framework. Additional ASN.1 macros may not be defined in SNMPv2 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.

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, even though hyphen is allowed by [1]. This restriction enables arithmetic expressions in languages which use the minus sign to reference these descriptors without ambiguity.)

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 Clauses

Some clauses in a macro invocation may take a textual value (e.g., the DESCRIPTION clause). Note that, in order to conform to the ASN.1 syntax, the entire value of these clauses must be enclosed in double quotation marks, and therefore cannot itself contain double quotation

marks, although the value may be multi-line.

### 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 even when there is no collision when IMPORTing symbols.

Finally, the IMPORTS statement may not be used to import an ASN.1 named type which corresponds to either the SEQUENCE or SEQUENCE OF type.

### 3.3. Exporting Symbols

The ASN.1 EXPORTS statement is not allowed in SNMPv2 information modules. All items defined in an information module are automatically exported.

### 3.4. ASN.1 Comments

Comments in ASN.1 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.

### 3.5. OBJECT IDENTIFIER values

An OBJECT IDENTIFIER value is an ordered list of non-negative numbers. For the SNMPv2 framework, 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
```

## 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 SNMPv2 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. The date and time are represented in UTC Time format (see Appendix B).

### 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. The date and time are represented in UTC Time format (see Appendix B).

#### 5.5.1. Mapping of the DESCRIPTION sub-clause

The DESCRIPTION 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.

## 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 "9505241811Z"
    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."
-- 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 values "current", and "obsolete" are self-explanatory. The "deprecated" value indicates that the definition is obsolete, but that an implementor may wish to support it to foster interoperability with older 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 an object assignment defined in some other information module.

### 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].

A extended subset of the full capabilities of ASN.1 sub-typing is allowed, as appropriate to the underingly ASN.1 type. Any such restriction on size, range, enumerations or repertoire 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 C 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 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 (no hyphens), 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.)

#### 7.1.2. OCTET STRING

The OCTET STRING type represents arbitrary binary or textual data. Although there is no SMI-specified size limitation for this type, 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 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.)

Although there is no SMI-specified limitation on the number of enumerations (and therefore on the length of a value), 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 (no hyphens), 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.)

#### 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 a SYNTAX clause value of TimeStamp (a textual convention defined in [3]) indicating the time of the last discontinuity.

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. The maximum value can not be greater than 2^32-1 (4294967295 decimal). The value of a Gauge has its maximum value whenever the information being modeled is greater or equal to that maximum value; if the information being modeled subsequently decreases below the maximum value, the Gauge also decreases.

### 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 a SYNTAX clause value of TimeStamp (a textual convention defined in [3]) indicating the time of the last discontinuity.

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, normally omitting the sub-typing information. Further, these ASN.1 types are always present (the 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 values "current", and "obsolete" are self-explanatory. The "deprecated" value indicates that the definition is obsolete, but that an implementor may wish to support that object to foster interoperability with older 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 an object defined in some other information module. This is useful when de-osifying a MIB module produced by some other organization.

### 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 syntax of those objects indicate how to form the instance-identifier:

(1) integer-valued: 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.

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 the SNMPv1 framework, and later converted to conform to the SNMPv2 framework; 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 1573's ifTable and the many media-specific MIBs which extend it for specific media (e.g., the dot3Table in RFC 1650), 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 1573's ifTable and a media-specific MIB which extends the ifTable for a specific media (e.g., the dot3Table in RFC 1650), 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 a SNMPv2 entity acting in an agent role when an object instance is created.

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 a SNMPv2 entity acting in an agent role might use.

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 a SNMPv2 entity acting in an agent role 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 }
  OBJECT IDENTIFIER  DEFVAL { sysDescr }
  BITS               DEFVAL { { primary, secondary } }
                     -- enumerated values that are set
  IpAddress          DEFVAL { 'c0210415'H } -- 192.33.4.21
```

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.

Further note that although conceptual tables and rows are given administratively assigned names, these conceptual objects may not be manipulated in aggregate form by the management protocol.

## 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      INTEGER
    MAX-ACCESS  read-only
    STATUS      current
    DESCRIPTION
            "The index number of the first unassigned entry in the
            evaluation table.

            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
    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 executed."
    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 the ordered sequence of MIB object types which are contained within every instance of the notification. An object type specified in this clause may not have an MAX-ACCESS clause of "not-accessible".

### 8.2. Mapping of the STATUS clause

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

The values "current", and "obsolete" are self-explanatory. The "deprecated" value indicates that the definition is obsolete, but that an implementor may wish to support the notification to foster interoperability with older 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 a notification defined in some other information module. This is useful when de-osifying a MIB module produced by some other organization.

### 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 the procedures employed by proxy agents (see Section 3.1.2 of [7]), 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 linkUp trap might be described:

```
linkUp NOTIFICATION-TYPE
    OBJECTS { ifIndex }
    STATUS  current
    DESCRIPTION
            "A linkUp trap signifies that the SNMPv2 entity, acting in
            an agent role, recognizes that one of the communication
            links represented in its configuration has come up."
    ::= { snmpTraps 4 }
```

According to this invocation, the trap authoritatively identified as

```
     { snmpTraps 4 }
```

is used to report a link coming up.

## 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 on | | | |
|---|---|---|---|---|
| object syntax | range | enumeration | size | repertoire |
| INTEGER | (1) | (2) | - | - |
| Integer32 | (1) | - | - | - |
| Unsigned32 | (1) | - | - | - |
| OCTET STRING | - | - | (3) | (4) |
| 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);

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

(4) the repertoire of characters in the value may be reduced by further sub-typing.

Otherwise no refinements are possible. Further details on sub-typing are provided in Appendix C.

## 10. Extending an Information Module

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

To begin, the invocation of the MODULE-IDENTITY macro should be updated to include information about the revision. Usually, this consists of updating the LAST-UPDATED clause and adding a pair of REVISION and DESCRIPTION clauses. However, other existing clauses in the invocation may be updated.

Note that the module's label (e.g., "FIZBIN-MIB" from the example in Section 5.8), is not changed when the information module is revised.

### 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.

(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".

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

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

(5) A UNITS clause may be added.

(6) A conceptual row may be augmented by adding new columnar objects at the end of the row.

(7) 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 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.

Finally, note that if an object has the value of its STATUS clause changed, then the value of its DESCRIPTION clause should be updated accordingly.

### 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.

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.

Finally, note that if an object has the value of its STATUS clause changed, then the value of its DESCRIPTION clause should be updated accordingly.

# 11. Appendix A: de-OSIfying a MIB module

There has been an increasing amount of work recently on taking MIBs defined by other organizations (e.g., the IEEE) and de-osifying them for use with the Internet-standard network management framework. The steps to achieve this are straight-forward, though tedious. Of course, it is helpful to already be experienced in writing MIB modules for use with the Internet-standard network management framework.

The first step is to construct a skeletal MIB module, as shown earlier in Section 5.8. The next step is to categorize the objects into groups. Optional objects are not permitted. Thus, when a MIB module is created, optional objects must be placed in a additional groups, which, if implemented, all objects in the group must be implemented. For the first pass, it is wisest to simply ignore any optional objects in the original MIB: experience shows it is better to define a core MIB module first, containing only essential objects; later, if experience demands, other objects can be added.

## 11.1. Managed Object Mapping

Next for each managed object class, determine whether there can exist multiple instances of that managed object class. If not, then for each of its attributes, use the OBJECT-TYPE macro to make an equivalent definition.

Otherwise, if multiple instances of the managed object class can exist, then define a conceptual table having conceptual rows each containing a columnar object for each of the managed object class's attributes. If the managed object class is contained within the containment tree of another managed object class, then the assignment of an object is normally required for each of the "distinguished attributes" of the containing managed object class. If they do not already exist within the MIB module, then they can be added via the definition of additional columnar objects in the conceptual row corresponding to the contained managed object class.

In defining a conceptual row, it is useful to consider the optimization of network management operations which will act upon its columnar objects. In particular, it is wisest to avoid defining more columnar objects within a conceptual row, than can fit in a single PDU. As a rule of thumb, a conceptual row should contain no more than approximately 20 objects. Similarly, or as a way to abide by the "20 object guideline", columnar objects should be grouped into tables according to the expected grouping of network management operations upon them. As such, the content of conceptual rows should reflect typical access scenarios, e.g., they should be organized

along functional lines such as one row for statistics and another row for parameters, or along usage lines such as commonly-needed objects versus rarely-needed objects.

On the other hand, the definition of conceptual rows where the number of columnar objects used as indexes outnumbers the number used to hold information, should also be avoided. In particular, the splitting of a managed object class's attributes into many conceptual tables should not be used as a way to obtain the same degree of flexibility/complexity as is often found in MIBs with a myriad of optionals.

#### 11.1.1. Mapping to the SYNTAX clause

When mapping to the SYNTAX clause of the OBJECT-TYPE macro:

(1) An object with BOOLEAN syntax becomes a TruthValue [3].

(2) An object with INTEGER syntax becomes an Integer32.

(3) An object with ENUMERATED syntax becomes an INTEGER with enumerations, taking any of the values given which can be represented with an Integer32.

(4) An object with BIT STRING syntax having enumerations becomes a BITS construct.

(5) An object with BIT STRING syntax but no enumerations becomes an OCTET STRING.

(6) An object with a character string syntax becomes either an OCTET STRING, or a DisplayString [3], depending on the repertoire of the character string.

(7) A non-tabular object with a complex syntax, such as REAL or EXTERNAL, must be decomposed, usually into an OCTET STRING (if sensible). As a rule, any object with a complicated syntax should be avoided.

(8) Tabular objects must be decomposed into rows of columnar objects.

#### 11.1.2. Mapping to the UNITS clause

If the description of this managed object defines a unit-basis, then mapping to this clause is straight-forward.

#### 11.1.3. Mapping to the MAX-ACCESS clause

This is straight-forward.

#### 11.1.4. Mapping to the STATUS clause

This is straight-forward.

#### 11.1.5. Mapping to the DESCRIPTION clause

This is straight-forward: simply copy the text, making sure that any embedded double quotation marks are sanitized (i.e., replaced with single-quotes or removed).

#### 11.1.6. Mapping to the REFERENCE clause

This is straight-forward: simply include a textual reference to the object being mapped, the document which defines the object, and perhaps a page number in the document.

#### 11.1.7. Mapping to the INDEX clause

If necessary, decide how instance-identifiers for columnar objects are to be formed and define this clause accordingly.

#### 11.1.8. Mapping to the DEFVAL clause

Decide if a meaningful default value can be assigned to the object being mapped, and if so, define the DEFVAL clause accordingly.

### 11.2. Action Mapping

Actions are modeled as read-write objects, in which writing a particular value results in a state change. (Usually, as a part of this state change, some action might take place.)

#### 11.2.1. Mapping to the SYNTAX clause

Usually the Integer32 syntax is used with a distinguished value provided for each action that the object provides access to. In addition, there is usually one other distinguished value, which is the one returned when the object is read.

#### 11.2.2. Mapping to the MAX-ACCESS clause

Always use read-write or read-create.

#### 11.2.3. Mapping to the STATUS clause

This is straight-forward.

#### 11.2.4. Mapping to the DESCRIPTION clause

This is straight-forward: simply copy the text, making sure that any embedded double quotation marks are sanitized (i.e., replaced with single-quotes or removed).

#### 11.2.5. Mapping to the REFERENCE clause

This is straight-forward: simply include a textual reference to the action being mapped, the document which defines the action, and perhaps a page number in the document.

### 11.3. Event Mapping

Events are modeled as SNMPv2 notifications using NOTIFICATION-TYPE macro. However, recall that SNMPv2 emphasizes trap-directed polling. As such, few, and usually no, notifications, need be defined for any MIB module.

#### 11.3.1. Mapping to the STATUS clause

This is straight-forward.

#### 11.3.2. Mapping to the DESCRIPTION clause

This is straight-forward: simply copy the text, making sure that any embedded double quotation marks are sanitized (i.e., replaced with single-quotes or removed).

#### 11.3.3. Mapping to the REFERENCE clause

This is straight-forward: simply include a textual reference to the notification being mapped, the document which defines the notification, and perhaps a page number in the document.

# 12. Appendix B: UTC Time Format

Several clauses defined in this document use the UTC Time format:

```
YYMMDDHHMMZ

where: YY - last two digits of year
       MM - month (01 through 12)
       DD - day of month (01 through 31)
       HH - hours (00 through 23)
       MM - minutes (00 through 59)
        Z - the character "Z" denotes Greenwich Mean Time (GMT).
```

For example, "9502192015Z" represents 8:15pm GMT on 19 February 1995.

# 13. Appendix C: Detailed Sub-typing Rules

## 13.1. 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 (i.e. 'xxxx'H)
    <binString> is a binary string (i.e. 'xxxx'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.

## 13.2. 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))
```

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
```

## 13.3. Rules for Textual Conventions

Sub-typing of Textual Conventions (see [3]) is allowed but must be valid. In particular, each range specified for the textual convention must be a subset of a range specified for the base type. For example,

```
Tc1 ::= INTEGER (1..10 | 11..20)
Tc2 ::= Tc1 (2..10 | 12..15)       -- is valid
Tc3 ::= Tc1 (4..8)                 -- is valid
Tc4 ::= Tc1 (8..12)                -- is invalid
```

# 14. Security Considerations

Security issues are not discussed in this memo.

## 15. Editor's Address

Keith McCloghrie
Cisco Systems, Inc.
170 West Tasman Drive
San Jose, CA 95134-1706
US

Phone: +1 408 526 5260
EMail: kzm@cisco.com

## 16. Acknowledgements

This document is the result of significant work by the four major contributors:

Jeffrey D. Case (SNMP Research, case@snmp.com)
Keith McCloghrie (Cisco Systems, kzm@cisco.com)
Marshall T. Rose (Dover Beach Consulting, mrose@dbc.mtview.ca.us)
Steven Waldbusser (International Network Services, stevew@uni.ins.com)

In addition, the contributions of the SNMPv2 Working Group are acknowledged. In particular, a special thanks is extended for the contributions of:

Alexander I. Alten (Novell)
Dave Arneson (Cabletron)
Uri Blumenthal (IBM)
Doug Book (Chipcom)
Kim Curran (Bell-Northern Research)
Jim Galvin (Trusted Information Systems)
Maria Greene (Ascom Timeplex)
Iain Hanson (Digital)
Dave Harrington (Cabletron)
Nguyen Hien (IBM)
Jeff Johnson (Cisco Systems)
Michael Kornegay (Object Quest)
Deirdre Kostick (AT&T Bell Labs)
David Levi (SNMP Research)
Daniel Mahoney (Cabletron)
Bob Natale (ACE*COMM)
Brian O'Keefe (Hewlett Packard)
Andrew Pearson (SNMP Research)
Dave Perkins (Peer Networks)
Randy Presuhn (Peer Networks)
Aleksey Romanov (Quality Quorum)
Shawn Routhier (Epilogue)
Jon Saperia (BGS Systems)

Bob Stewart (Cisco Systems, bstewart@cisco.com), chair
Kaj Tesink (Bellcore)
Glenn Waters (Bell-Northern Research)
Bert Wijnen (IBM)

## 17. 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] SNMPv2 Working Group, Case, J., McCloghrie, K., Rose, M., and S. Waldbusser, "Conformance Statements for Version 2 of the Simple Network Management Protocol (SNMPv2)", RFC 1904, January 1996.

[3] SNMPv2 Working Group, Case, J., McCloghrie, K., Rose, M., and S. Waldbusser, "Textual Conventions for Version 2 of the Simple Network Management Protocol (SNMPv2)", RFC 1903, January 1996.

[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] 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] 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.

[7] SNMPv2 Working Group, Case, J., McCloghrie, K., Rose, M., and S. Waldbusser, "Coexistence between Version 1 and Version 2 of the Internet-standard Network Management Framework", RFC 1908, January 1996.