HEMS Monitoring and Control Language

TABLE OF CONTENTS

1. STATUS OF THIS MEMO

   This RFC specifies a query language for monitoring and control of
   network entities. This RFC supercedes RFC-1023, extending the query
   language and providing more discussion of the underlying issues.

This language is a component of the High-Level Entity Monitoring
System (HEMS) described in RFC-1021 and RFC-1022. Readers may wish
to consult these RFCs when reading this memo. RFC-1024 contains
detailed assignments of numbers and structures used in this system.
Portions of RFC-1024 that define query language structures are
superceded by definitions in this memo. This memo assumes a
knowledge of the ISO data encoding standard, ASN.1.

Distribution of this memo is unlimited.

INTRODUCTION

This RFC specifies the design of a general-purpose, yet efficient,
monitoring and control language for managing network entities. The
data in the entity is modeled as a hierarchy and specific items are
named by giving the path from the root of the tree. Most items are
read-only, but some can be "set" in order to perform control
operations. Both requests and responses are represented using the
ISO ASN.1 data encoding rules.

2. OVERVIEW AND SCOPE

The basic model of monitoring and control used in this memo is that a
query is sent to a monitored entity and the entity sends back a
response. The term query is used in the database sense -- it may
request information, modify data, or both. We will use gateway-
oriented examples, but it should be understood that this query-
response mechanism is applicable to any IP entity.

In particular, there is no notion of an interactive "conversation" as
in SMTP [RFC-821] or FTP [RFC-959]. A query is a complete request
that stands on its own and elicits a complete response.

In order to design the query language, we had to define a model for
the data to be retrieved by the queries, which required some
understanding of and assumptions to be made about the data. We ended
up with a fairly flexible data model, which places few limits on the
type or size of the data.

Wherever possible, we give motivations for the design decisions or
assumptions that led to particular features or definitions. Some of
the important global considerations and assumptions are:

        - The query processor should place as little computational
          burden on the monitored entity as possible.

        - It should not be necessary for a monitored entity to store
          the complete query. Nothing in the query language should

preclude an implementation from being able to process the
query on the fly, producing portions of the response while
the query is still being read and parsed. There may be
other constraints that require large amounts of data to be
buffered, but the query language design must not be one.

- It is assumed that there is some mechanism to transport a
  sequence of octets to a query processor within the
  monitored entity and that there is some mechanism to return
  a sequence of octets to the entity making the query. In
  HEMS, this is provided by HEMP and its underlying transport
  layer. The query language design is independent of these
  details, however, and could be grafted onto some other
  protocol.

- The data model must provide organization for the data, so
  that it can be conveniently named.

- Much of the data to be monitored will be contained in
  tables. Some tables may contain other tables. The query
  language should be able to deal with such tables.

- We don't provide capabilities for data reduction in the
  query language. We will provide for data selection, for
  example, only retrieving certain table entries, but we will
  not provide general facilities for processing data, such as
  computing averages.

- Because one monitoring center may be querying many
  (possibly hetrogenous) hosts, it must be possible to write
  generic queries that can be sent to all hosts, and have the
  query elicit as much information as is available from each
  host. i.e., queries must not be aborted just because they
  requested non-existent data.

There were some assumptions that we specifically did not make:

- It is up to the implementation to choose what degree of
  concurrency will be allowed when processing queries. By
  locking only portions of the database, it should be
  possible to achieve good concurrency while still preventing
  deadlock.

- This specification makes no statement about the use of the
  "definite" and "indefinite" length forms in ASN.1. There
  is currently some debate about this usage in the ISO
  community; implementors should note the recommendations in
  the ASN.1 specification.

Other RFCs associated with HEMS are:

    RFC-1021   Overview;
    RFC-1022   Transport protocol and message encapsulation;
    RFC-1024   Precise data definitions.

The rest of this report is organized as follows:

    Section 3   Gives a brief overview of the data model and the
                      operation of the query processor.

    Section 4   Describes the encoding used for queries and
                      responses, and the notation used to represent them
                      in this report.

    Section 5   Describes how the data is organized in the
                      monitored entity, and the view provided of it by
                      the query processor.

    Section 6   Describes the basic data types that may be given
                      to the query processor as input.

    Section 7   Describes how a reply to a query is organized.

    Section 8   Describes the operations available in the query
                      language.

    Section 9   Describes how the set of data in the tree may be
                      extended.

    Section 10   Describes how authorization issues affect the
                      execution of a query.

    Section 11   Describes how errors are reported, and their
                      effect on the processing of the query.

    Appendix I   Gives precise ASN.1 definitions of the data types
                      used by the query processor.

    Appendix II   Gives extensive implementation hints for the core
                      of the query processor.

3. OVERVIEW OF QUERY PROCESSOR OPERATION

    In this section, we give an overview of the operation of the query
    processor, to provide a framework for the later sections.

    The query language models the manageable data as a tree, with each

branch representing a different aspect of the entity, such as
different layers of protocols. Subtrees are further divided to
provide additional structure to the data. The leaves of the tree
contain the actual data.

Given this data representation, the task of the query processor is to
traverse this tree and retrieve (or modify) data specified in a
query. A query consists of instructions to move around in the tree
and to retrieve (or modify) named data. The result of a query is an
exact image of the parts of the tree that the query processor
visited.

The query processor is very simple -- it only understands eight
commands, most of which share the same structure. It is helpful to
think of the query processor as an automaton that walks around in the
tree, directed by commands in the query. As it moves around, it
copies the tree structure it traverses to the query result. Data
that is requested by the query is copied into the result as well.
Data that is changed by a query is copied into the result after the
modification is made.

4. ENCODING OF QUERIES AND RESPONSES

Both queries and responses are encoded using the representation
defined in ISO Standard ASN.1 (Abstract Syntax Notation 1). ASN.1
represents data as sequences of <tag,length,contents> triples that
are encoded as a stream of octets. The data tuples may be
recursively nested to represent structured data such as arrays or
records. For a full description, see the ISO standards IS 8824 and
IS 8825. See appendix for information about obtaining these
documents.

4.1 Notation Used in This Proposal

The notation used in this memo is similar to that used in ASN.1, but
less formal, smaller, and (hopefully) easier to read. We will refer
to a <tag,length,contents> tuple as a "data object". In this RFC, we
will not be concerned with the details of the object lengths. They
exist in the actual ASN.1 encoding, but will be omitted in the
examples here.

Data objects that have no internal ASN.1 structure such as integer or
octet string are referred to as "simple types" or "simple objects".
Objects which are constructed out of other ASN.1 data objects will be
referred to as "composite types" or "composite objects".

The notation
     ID(value)
represents a simple object whose tag is "ID" with the given value. A
composite object is represented as
     ID{ ... contents ... }
where contents is a sequence of data objects. The contents may
include both simple and structured types, so the structure is fully
recursive.

The difference between simple and composite types is close to the
meaning of the "constructor" bit in ASN.1. For the uses here, the
distinction is made based upon the semantics of the data, not the
representation. Therefore, even though an OctetString can be
represented in ASN.1 using either constructed or non-constructed
forms, it is conceptually a simple type, with no internal structure,
and will always be written as
     ID("some arbitrary string")
in this RFC.

There are situations where it is necessary to specify a type but give
no value, such as when referring to the name of the data. In this
situation, the same notation is used, but with the value omitted:
     ID   or ID()   or   ID{}
Such objects have zero length and no contents. The latter two forms
are used when a distinction is being made between simple and
composite data, but the difference is just notation -- the
representation is the same.

ASN.1 distinguishes between four "classes" of tags: universal,
application-specific, context-dependent, and reserved. HEMS and this
query language use the first three. Universal tags are assigned in
the ASN.1 standard and its addendums for common types, and are
understood by any application using ASN.1. Application-specific tags
are limited in scope to a particular application. These are used for
"well-known" identifiers that must be recognizable in any context,
such as derived data types. Finally, context-dependent tags are used
for objects whose meaning is dependent upon where they are
encountered. Most tags that identify data are context-dependent.

5. DATA ORGANIZATION

   Data in a monitored entity is modeled as a hierarchy.
   Implementations are not required to organize the data internally as a
   hierarchy, but they must provide this view of the data through the
   query language. A hierarchy offers useful structure for the
   following operations:

Organization   A hierarchy allows related data to be grouped
                  together in a natural way.

Naming   The name of a piece of data is just the path from the
                  root to the data of interest.

Mapping onto ASN.1
                  ASN.1 can easily represent a hierarchy by using a
                  "constructor" type as an envelope for an entire
                  subtree.

Efficient Representation
                  Hierarchical structures are compact and can be
                  traversed quickly.

Safe Locking   If it is necessary to lock part of the hierarchy (for
                  example, when doing an update), locking an entire
                  subtree can be done efficiently and safely, with no
                  danger of deadlock.

We will use the term "data tree" to refer to this entire structure.
Note that this internal model is completely independent of the
external ASN.1 representation -- any other suitable representation
would do. For the sake of efficiency, we do make a one-to-one
mapping between ASN.1 tags and the (internal) names of the nodes.
The same could be done for any other external representation.

Each node in the hierarchy must have names for its component parts.
Although we would normally think of names as being ASCII strings such
as "input errors", the actual name is just an ASN.1 tag. Such names
are small integers (typically, less than 30) and so can easily be
mapped by the monitored entity onto its internal representation.

We use the term "dictionary" to mean an internal node in the
hierarchy. Leaf nodes contain the actual data. A dictionary may
contain both leaf nodes and other dictionaries.

5.1 Example Data Tree

Here is a possible organization of the hierarchy in an entity that
has several network interfaces and does IP routing. The exact
organization of data in entities is specified in RFC-1024. This
skeletal data tree will be used throughout this RFC in query
examples.

        System {
                name   -- host name
                clock-msec   -- msec since boot

```
                interfaces                      -- # of interfaces
                memory
                }
        Interfaces {                            -- one per interface
                InterfaceData{ address, mtu, netMask, ARP{...}, ... }
                InterfaceData{ address, mtu, netMask, ARP{...}, ... }
                        :
                }
        IPRouting {
                Entry{ ip-addr, interface, cost, ... }
                Entry{ ip-addr, interface, cost, ... }
                        :
                }
```

There are three top-level dictionaries in this hierarchy (System,
Interfaces, and IPRouting) and three other dictionary types
(InterfaceData, Entry, and ARP), each with multiple instances.

The "name" of the clock in this entity would be:
    system{ clock-msec }
and the name of a routing table entry's IP address would be:
    IPRouting{ Entry{ ip-addr } }.

More than one piece of data can be named by a single ASN.1 object.
The entire collection of system information is named by:
    system
and the name of a routing table's IP address and cost would be:
    IPRouting{ Entry{ ip-addr, cost } }.

5.2 Arrays

  There is one sub-type of a dictionary that is used as the basis for
  tables of objects with identical types. We call these dictionaries
  arrays. In the example above, the dictionaries for interfaces,
  routing tables, and ARP tables are all arrays.

  In the examples above, the "ip-addr" and "cost" fields are named. In
  fact, these names refer to the field values for ALL of the routing
  table entries -- the name doesn't (and can't) specify which routing
  table entry is intended. This ambiguity is a problem wherever data
  is organized in tables. If there was a meaningful index for such
  tables (e.g., "routing table entry #1"), there would be no problem.
  Unfortunately, there usually isn't such an index. The solution to
  this problem requires that the data be accessed on the basis of some
  of its content. Filters, discussed in section 8.6, provide this
  mechanism.

  The primary difference between arrays and plain dictionaries is that

arrays may contain only one type of item, while dictionaries, in
general, will contain many different types of items. For example,
the dictionary IPRouting (which is an array) will contain only items
of type Entry.

The fact that these objects are viewed externally as arrays or tables
does not mean that they are represented in an implementation as
linear lists of objects. Any collection of same-typed objects is
viewed as an array, even though it might be stored internally in some
other format, for example, as a hash table.

6. COMPONENTS OF A QUERY

A HEMS query consists of a sequence of ASN.1 objects, interpreted by
a simple stack-based interpreter. [Although we define the query
language in terms of the operations of a stack machine, the language
does not require an implementation to use a stack machine. This is a
well-understood model, and is easy to implement.] One ASN.1 tag is
reserved for operation codes; all other tags indicate data that will
eventually be used by an operation. These objects are pushed onto
the stack when received. Opcodes are immediately executed and may
remove or add items to the stack. Because ASN.1 itself provides
tags, very little needs to be done to the incoming ASN.1 objects to
make them suitable for use by the query interpreter.

Each ASN.1 object in a query will fit into one of the following
categories:

Opcode   An opcode tells the query interpreter to perform an action.
          They are described in detail in section 8. Opcodes are
          represented by an application-specific type whose value
          determines the operation.

Template These are objects that name one or more items in the data
          tree. Named items may be either simple items (leaf nodes)
          or entire dictionaries, in which case the entire subtree
          "underneath" the dictionary is understood. Templates are
          used to select specific data to be retrieved from the data
          tree. A template may be either simple or structured,
          depending upon what it is naming. A template only names
          the data -- there are no values contained in it. Therefore
          the leaf objects in a template will all have a length of
          zero.

          Examples of very simple templates are:
              name()   or   System{}
          Each of these is just one ASN.1 data object, with zero
          length. The first names a single data item in the "System"

dictionary (and must appear in that context), and the
second names the entire "System" dictionary. A more
complex template such as:
          Interfaces{ InterfaceData{ address, netMask, ARP } }
names two simple data items and a dictionary, iterated over
all occurrences of "InterfaceData" within the Interfaces
array.

Path   A path is a special case of a template that names only a
           single node in the tree. It specifies a path down into the
           dictionary tree and names exactly one node in the
           dictionary tree.

Value   These are used to give data values when needed in a query,
           for example, when changing a value in the data tree. A
           value can be thought of as either a filled-in template or
           as the ASN.1 representation some part of the data tree.

Filter   A boolean expression that can be executed in the context of
           a particular dictionary that is used to select or not
           select items in the dictionary. The expressions consist of
           the primitives "equal", "greater-or-equal",
           "less-or-equal", and "present" possibly joined by "and",
           "or", and "not". (See section 8.6.)

Values, Paths, and Templates usually have names in the context-
dependent class, except for a few special cases, which are in the
application-specific class.

7. REPLY TO A QUERY

The data returned to the monitoring entity is a sequence of ASN.1
data items. Conceptually, the reply is a subset of the data tree,
where the query selects which portions are to be included. This is
exactly true for data retrieval requests, and essentially true for
data modification requests -- the reply contains the data after it
has been modified. The key point is that the data in a reply
represents the state of the data tree immediately after the query was
executed.

The sequence of the data is determined by the sequence of query
language operations and the order of data items within Templates and
Values given as input to these operations. If a query requests data
from two of the top-level dictionaries in the data tree, by giving
two templates such as:

          System{ name, interfaces }
          Interfaces{

```
                    InterfaceData { address, netMask, mtu }
                    }
```

   then the response will consist of two ASN.1 data objects, as follows:

```
        System {
                name("system name"),
                interfaces(2)
                }
        Interfaces {
                InterfaceData { address(36.8.0.1),
                                netMask(FFFF0000),
                                mtu(1500)
                                }
                InterfaceData { address(10.1.0.1),
                                mtu(1008),
                                netMask(FF000000)
                                }
                }
```

   With few exceptions, each of the data items in the hierarchy is named
   in the context-specific ASN.1 type space. Because of this, the
   returned objects must be fully qualified. For example, the name of
   the entity must always be returned encapsulated inside an ASN.1
   object for "System". If it were not, there would be no way to tell
   if the object that was returned was "name" inside the "System"
   dictionary or "address" inside the "interfaces" dictionary (assuming
   in this case that "name" and "address" were assigned the same integer
   as their ASN.1 tags).

   Having fully-qualified data simplifies decoding of the data at the
   receiving end and allows the tags to be locally chosen. Definitions
   for tags within routing tables won't conflict with definitions for
   tags within interfaces. Therefore, the people doing the name
   assignments are less constrained. In addition, most of the
   identifiers will be fairly small integers, which is an advantage
   because ASN.1 can fit tag numbers up to 30 in a one-octet tag field.
   Larger numbers require a second octet.

   If data is requested that doesn't exist, either because the tag is
   not defined, or because an implementation doesn't provide that data
   (such as when the data is optional), the response will contain an
   ASN.1 object that is empty. The tag will be the same as in the
   query, and the object will have a length of zero.

   The same response is given if the requested data does exist, but the
   invoker of the query does not have authorization to access it. See
   section 10 for more discussion of authorization mechanisms.

This allows completely generic queries to be composed without regard
to whether the data is defined or implemented at all of the entities
that will receive the query. All of the available data will be
returned, without generating errors that might otherwise terminate
the processing of the query.

8. QUERY LANGUAGE

The query language is designed to be expressive enough to write
useful queries with, yet simple enough to be easy to implement. The
query processor should be as simple and fast as possible, in order to
avoid placing a burden on the monitored entity, which may be a
critical node such as a gateway.

Although queries are formed in a flexible way using what we term a
"language", this is not a programming language. There are operations
that operate on data, but most other features of programming
languages are not present. In particular:

        - Programs are not stored in the query processor.

        - The only form of temporary storage is a stack, of limited
          depth.

        - There are no subroutines.

        - There are no explicit control structures defined in the
          language.

The central element of the language is the stack. It may contain
templates, (and therefore paths), values, and filters taken from the
query. In addition, it can contain dictionaries (and therefore
arrays) from the data tree. At the beginning of a query, it contains
one item, the root dictionary.

The overall operation consists of reading ASN.1 objects from the
input stream. All objects that aren't opcodes are pushed onto the
stack as soon as they are read. Each opcode is executed immediately
and may remove items from the stack, may generate ASN.1 objects and
send them to the output stream, and may leave items on the stack.
Because each input object is dealt with immediately, portions of the
response may be generated while the query is still being received.

In the descriptions below, operator names are in capital letters,
preceded by the arguments used from the stack and followed by results
left on the stack. For example:

```
    OP                           a b  OP   a t
            means that the OP operator takes <a> and <b> off of the
            stack and leaves <t> on the stack.  Most of the operators
            in the query language leave the first operand (<a> in this
            example) on the stack for future use.
```

If both <a> and <b> were received as part of the query (as opposed to
being calculated by previous operations), then this part of the query
would have consisted of the sequence:
```
    <a>
    <b>
    OP
```
So, like other stack-based languages, the arguments and operators
must be presented in postfix order, with an operator following its
operands.

Here is a summary of all of the operators defined in the query
language. Most of the operators can take several different sets of
operands and behave differently based upon the operand types.
Details and examples are given later.

```
    BEGIN                  dict1 path  BEGIN   dict1 dict
                   array path filter  BEGIN   array dict
            Move down in the data tree, establishing a context for
            future operations.

    END                          dict  END   --
            Undo the most recent BEGIN.

    GET                          dict  GET   dict
                       dict template  GET   dict
               array template filter  GET   array
            Retrieve data from the data tree.

    GET-ATTRIBUTES
                             dict  GET-ATTRIBUTES   dict
                   dict template  GET-ATTRIBUTES   dict
           array template filter  GET-ATTRIBUTES   array
            Retrieve attribute information about data in the data tree.

    GET-RANGE   dict path start length  GET-RANGE   dict
            Retrieve a subrange of an OctetString.  Used for reading
            memory.

    SET                     dict value  SET   dict
                    array value filter  SET   array
            Change values in the data tree, possibly performing control
            functions.
```

```
     CREATE                   array value   CREATE    dict
            Create new table entries.

     DELETE                   array filter  DELETE    array
            Delete table entries.
```

These operators are defined so that it is impossible to generate an
invalid query response. Since a response is supposed to be a
snapshot of a portion (or portions) of the data tree, it is important
that only data that is actually in the tree be put in the response.
Two features of the language help guarantee this:

    - Data is put in the response directly from the tree (by
      GET-*). Data does not go from the tree to the stack and
      then into the response.

    - Dictionaries on the stack are all derived from the initial,
      root dictionary. The operations that manipulate
      dictionaries (BEGIN and END) also update the response with
      the new location in the tree.

8.1 Moving Around in the Data Tree

   The initial point of reference in the data tree is the root. That
   is, operators name data starting at the root of the tree. It is
   useful to be able to move to some other dictionary in the tree and
   then name data from that point. The BEGIN operator moves down in the
   tree and END undoes the last unmatched BEGIN.

   BEGIN is used for two purposes:

      - By moving to a dictionary closer to the data of interest,
        the name of the data can be shorter than if the full name
        (from the root) were given.

      - It is used to establish a context for filtered operations
        to operate in. Filters are discussed in section 8.6.

     BEGIN   dict1 path BEGIN   dict1 dict
            Follow <path> down the dictionary starting from <dict1>.
            Push the final dictionary named by <path> onto the stack.
            <path> must name a dictionary (not a leaf node). At the
            same time, produce the beginning octets of an ASN.1 object
            corresponding to the new dictionary. It is up to the
            implementation to choose between using the "indefinite
            length" representation or the "definite length" form and
            going back and filling the length in later.

        END   dict   END   --
                Pop <dict> off of the stack and terminate the open ASN.1
                object(s) started by the matching BEGIN. Must be paired
                with a BEGIN. If an END operation pops the root dictionary
                off of the stack, the query is terminated.

     <path> must point to a regular dictionary. If any part of it refers
     to a non-existent node, if it points to a leaf node, or if it refers
     to a node inside an array-type dictionary, then it is in error, and
     the query is terminated immediately.

     An additional form of BEGIN, which takes a filter argument, is
     described later.

8.2 Retrieving Data

     The basic model that all of the data retrieval operations follow is
     that they take a template and fill in the leaf nodes of the template
     with the appropriate data values.

     GET   dict template   GET   dict
                Emit an ASN.1 object with the same "shape" as the given
                template, except with values filled in for each node. The
                first ASN.1 tag of <template> should refer to an object in
                <dict>. If a dictionary tag is supplied anywhere in
                <template>, the entire dictionary contents are emitted to
                the response. Any items in the template that are not in
                <dictionary> (or its components) are represented as objects
                with a length of zero.

                               dict   GET   dict
                If there is no template, get all of the items in the
                dictionary. This is equivalent to providing a template
                that lists all of the items in the dictionary.

     An additional form of GET, which takes a filter argument, is
     described later.

     Here is an example of using the BEGIN operator to move down the data
     tree to the TCP dictionary and then using the GET operator to
     retrieve 5 data values from the TCP Stats dictionary:

        IPTransport{ TCP } BEGIN
        Stats{ octetsIn, octetsOut, inputPkts, outputPkts, badtag } GET
        END

This might return:

```
IPTransport{ TCP
    Stats{ octetsIn(13255), octetsOut(82323),
           inputPkts(9213), outputPkts(12425), badtag() }
}
```

"badtag" is a tag value that is undefined. No value is returned for
it, indicating that there is no data value associated with it.

8.3 Data Attributes

Although ASN.1 "self-describes" the structure and syntax of the data,
it gives no information about what the data means. For example, by
looking at the raw data, it is possible to tell that an item is of
type [context 5] and is 4 octets long. That does not tell how to
interpret the data (is this an integer, an IP address, or a 4-
character string?) or what the data means (IP address of what?).
Even if the data were "tagged", in ASN.1 parlance, that would only
give the base type (e.g., IP-address or counter) and not the meaning.

Most of the time, this information will come from RFC-1024, which
defines the ASN.1 tags and their precise meaning. When extensions
have been made, it may not be possible to get documentation on the
extensions. (Extensions are discussed in section 9.)

The GET-ATTRIBUTES operator is similar to the GET operator, but
returns a set of attributes describing the data rather than the data
itself. This information is intended to be sufficient to let a human
understand the meaning of the data and to let a sophisticated
application treat the data appropriately. Such an application could
use the attribute information to format the data on a display and
decide whether it is appropriate to subtract one sample from another.

Some of the attributes are textual descriptions to help a human
understand the nature of the data and provide meaningful labels for
it. Extensive descriptions of standard data are optional, since they
are defined in RFC-1024. Complete descriptions of extensions must be
available, even if they are documented in a user's manual. Network
firefighters may not have a current manual handy when the network is
broken.

The format of the attributes is not as simple as the format of the
data itself. It isn't possible to use the data's tag, since that
would look exactly like the data itself. The format is:

```
Attributes ::= [APPLICATION 3] IMPLICIT SEQUENCE {
        tagASN1          [0] IMPLICIT INTEGER,
```

```
                valueFormat      [1] IMPLICIT INTEGER,
                longDesc         [2] IMPLICIT IA5String OPTIONAL,
                shortDesc        [3] IMPLICIT IA5String OPTIONAL,
                unitsDesc        [4] IMPLICIT IA5String OPTIONAL,
                precision        [5] IMPLICIT INTEGER OPTIONAL,
                properties       [6] IMPLICIT BITSTRING OPTIONAL,
                valueSet         [7] IMPLICIT SET OF valueDesc OPTIONAL
                }
```

   The GET-ATTRIBUTES operator is similar to the GET operator. The
   major difference is that dictionaries named in the template do not
   elicit data for the entire subtree.

   GET-ATTRIBUTES
                     dict template GET-ATTRIBUTES   dict
             Emit a single ASN.1 Attributes object for each of the
             objects named in <template>. For each of these, the
             tagASN1 field will be set to the corresponding tag from the
             template. The rest of the fields are set as appropriate
             for the data object. Any items in the template that are
             not in <dictionary> (or its components) elicit an
             Attributes object with a valueFormat of NULL, and no other
             descriptive information.

      or
                       dict GET-ATTRIBUTES   dict
             If there is no template, emit Attribute objects for all of
             the items in the dictionary. This is equivalent to
             providing a template that lists all of the items in the
             dictionary. This allows a complete list of a dictionary's
             contents to be obtained.

   An additional form of GET-ATTRIBUTES, which takes a filter argument,
   is described later.

   Here is an example of using the GET-ATTRIBUTES operator to request
   attributes for three objects in the System dictionary:

        System{ name, badtag, clock-msec } GET-ATTRIBUTES

   "badtag" is some unknown tag. The result might be:

        System{
              Attributes{
                      tagASN1(name),
                      valueFormat(IA5String),
                      longDesc("The primary hostname."),
```

```
                        shortDesc("hostname")
                },
                Attributes{
                        tagASN1(badtag), valueFormat(NULL)
                }
                Attributes{
                        tagASN1(clock-msec),
                        valueFormat(Integer),
                        longDesc("milliseconds since boot"),
                        shortDesc("uptime"), unitsDesc("ms")
                        precision(4294967296),
                        properties(1)
                }
```

   Note that in this example "name" and "clock-msec" are integer values
   for the ASN.1 tags for the two data items. "badtag" is an integer
   value that has no corresponding name in this context.

   There will always be exactly as many Attributes items in the result
   as there are objects in the template. Attributes objects for items
   which do not exist in the entity will have a valueFormat of NULL and
   none of the optional elements will appear.

      [A much cleaner method would be to store the attributes as
      sub-components of the data item of interest. For example,
      requesting
          System{ clock-msec } GET
      would normally just get the value of the data. Asking for an
      additional layer down the tree would now get its attributes:
          System{ clock-msec{ shortDesc, unitsDesc } GET
      would get the named attributes. (The attributes would be
      named with application-specific tags.) Unfortunately, ASN.1
      doesn't provide a notation to describe this type of
      organization. So, we're stuck with the GET-ATTRIBUTES
      operator. However, if a cleaner organization were possible,
      this decision would have been made differently.]

8.4 Examining Memory

   Even with the ability to symbolically access all of this information
   in an entity, there will still be times when it is necessary to get
   to very low levels and examine memory, as in remote debugging
   operations. The building blocks outlined so far can easily be
   extended to allow memory to be examined.

   Memory is modeled as an ordinary object in the data tree, with an
   ASN.1 representation of OctetString. Because of the variety of
   addressing architectures in existence, the conversion from the

internal memory model to OctetString is very machine-dependent. The
only simple case is for byte-addressed machines with 8 bits per byte.

Each address space in an entity is represented by one "memory" data
item. In a one-address-space situation, this dictionary will
probably be in "System" dictionary. If each process has its own
address space, then one "memory" item might exist for each process.
Again, this is very machine-dependent.

Although the GET-RANGE operator is provided primarily for the purpose
of retrieving the contents of memory, it can be used on any object
whose base type is OctetString.

GET-RANGE   dict path start length   GET-RANGE   dict
          Get <length> elements of the OctetString, starting at
          <start>. <start> and <length> are both ASN.1 INTEGER type.
          <path>, starting from <dict>, must specify a node
          representing memory, or some other OctetString.

The returned data may not be <length> octets long, since it may take
more than one octet to represent one memory location.

Memory items in the data tree are special in that they will not
automatically be returned when the entire contents of a dictionary
are requested. e.g., If memory is part of the "System" dictionary,
then the query
     System GET
will emit the entire contents of the System dictionary, but not the
memory item.

8.5 Control Operations: Modifying the Data Tree

All of the operators defined so far only allow data in an entity to
be retrieved. By replacing the template argument used in the GET
operators with a value, data in the entity can be changed. Very few
items in the data tree can be changed; those that can are noted in
RFC-1024.

Values in the data tree can modified in order to change configuration
parameters, patch routing tables, etc. Control functions, such as
bringing an interface "down" or "up", do not usually map directly to
changing a value. In such cases, an item in the tree can be defined
to have arbitrary side-effects when a value is assigned to it.
Control operations then consist of "setting" this item to an
appropriate command code. Reading the value of such an item might
return the current status. Again, details of such data tree items
are given in RFC-1024.

This "virtual command-and-status register" model is very powerful, and can be extended by an implementation to provide whatever controls are needed. It has the advantage that the control function is associated with the controlled object in the data tree. In addition, no additional language features are required to support control functions, and the same operations used to locate data for retrieval are used to describe what is being controlled.

For all of the control and data modification operations, the fill-in-the-blank model used for data retrieval is extended: the response to an operation is the affected part of the data tree, after the operation has been executed. Therefore, for normal execution, SET and CREATE will return the object given as an argument, and DELETE will return nothing (because the affected portion was deleted).

SET   dict value   SET   dict
          Set the value(s) of data in the entity to the value(s)
          given in <value>. The result will be the value of the data
          after the SET. Attempting to set a non-settable item will
          not produce an error, but will yield a result in the reply
          different from what was sent.

CREATE   array value   CREATE   dict
          Insert a new entry into <array>. Depending upon the
          context, there may be severe restrictions about what
          constitutes a valid <value>. The result will be the actual
          item added to the <array>. Note that only one item can be
          added per CREATE operation.

DELETE   array filter   DELETE   array
          Delete the entry(s) in <array> that match <filter>.
          Filters are described later in this document. Normally, no
          data items will be produced in the response, but if any of
          the items that matched the filter could not be deleted,
          they will be returned in the response.

An additional form of SET, which takes a filter argument, is described later.

Here is an example of attempting to use SET to change the number of interfaces in an entity:
    System{ interfaces(5) } SET
Since that is not a settable parameter, the result would be:
    System{ interfaces(2) }
giving the old value.

Here is an example of how CREATE would be used to add a routing table entry for net for 128.89.0.0.

```
       IPRouting BEGIN                    -- get dictionary
       Entries{ DestAddr(128.89.0.0), ... }    -- entry to insert
       CREATE
       END                                -- finished with dict
```

   The result would be what was added:
```
       IPRouting{ Entries{ DestAddr(128.89.0.0), ... } }
```

   The results in the response of these operators is consistent of the
   global model of the response: it contains a subset of the data in
   the tree immediately after the query is executed.

   Note that CREATE and DELETE only operate on arrays, and then only on
   arrays that are specifically intended for it. For example, it is
   quite reasonable to add and remove entries from routing tables or ARP
   tables, both of which are arrays. However, it doesn't make sense to
   add or remove entries in the "Interfaces" dictionary, since the
   contents of that array is dictated by the hardware. For each array
   in the data tree, RFC-1024 indicates whether CREATE and DELETE are
   valid.

   CREATE and DELETE are always invalid in non-array contexts. If
   DELETE were allowed on monitored data, then the deleted data would
   become unmonitorable to the entire world. Conversely, if it were
   possible to CREATE arbitrary dictionary entries, there would be no
   way to give such entries any meaning. Even with the data in place,
   there is nothing that would couple the data to the operation of the
   monitored entity. [Creation and deletion would also add considerable
   complication to an implementation, because without them, all of the
   data structures that represent the data tree are essentially static,
   with the exception of dynamic tables such as the ones mentioned,
   which already have mechanisms in place for adding and removing
   entries.]

8.6 Associative Data Access: Filters

   One problem that has not been dealt with was alluded to earlier: When
   dealing with array data, how do you specify one or more entries based
   upon some value in the array entries? Consider the situation where
   there are several interfaces. The data might be organized as:

```
       Interfaces {                          -- one per interface
              InterfaceData { address, mtu, netMask, ARP{...}, ... }
              InterfaceData { address, mtu, netMask, ARP{...}, ... }
                              :
              }
```

   If you only want information about one interface (perhaps because

there is an enormous amount of data about each), then you have to
have some way to name it. One possibility would be to just number
the interfaces and refer to the desired interface as
    InterfaceData(3)
for the third one.

But this is not sufficient, because interface numbers may change over
time, perhaps from one reboot to the next. It is even worse when
dealing with arrays with many elements, such as routing tables, TCP
connections, etc. Large, changing arrays are probably the more
common case, in fact. Because of the lack of utility of indexing in
this context, there is no general mechanism provided in the language
for indexing.

A better scheme is to select objects based upon some value contained
in them, such as the IP address. The query language uses filters to
select specific table entries that an operator will operate on. The
operators BEGIN, GET, GET-ATTRIBUTES, SET, and DELETE can take a
filter argument that restricts their operation to entries that match
the filter.

A filter is a boolean expression that is executed for each element in
an array. If an array entry "matches" the filter (i.e., if the filter
produces a "true" result), then it is used by the operation. A
filter expression is very restricted: it can only compare data
contained in the array element and the comparisons are only against
constants. Comparisons may be connected by AND, OR and NOT
operators.

The ASN.1 definition of a filter is:

```
    Filter              ::= [APPLICATION 2] CHOICE {
                            present         [0] DataPath,
                            equal           [1] DataValue,
                            greaterOrEqual  [2] DataValue,
                            lessOrEqual     [3] DataValue,
                            and             [4] SEQUENCE OF Filter,
                            or              [5] SEQUENCE OF Filter,
                            not             [6] Filter
                            }

    DataPath        ::= ANY                 -- Path with no value

    DataValue       ::= ANY                 -- Single data value
```

This definition is similar to the filters used in the ISO monitoring
protocol (CMIP) and was derived from that specification.

"DataPath" is the name of a single data item; "DataValue" is the
value of a single data item. The three comparisons are all of the
form "data OP constant", where "data" is the value from the tree,
"constant" is the value from the filter expression, and "OP" is one
of equal, greater-than-or-equal, or less-than-or-equal. The last
operation, "present", tests to see if the named item exists in the
data tree. By its nature, it requires no value, so only a path needs
to be given.

Here is an example of a filter that matches an Interface whose IP
address is 10.1.0.1:
    Filter{ equal{ address(10.0.0.51) } }
Note that the name of the data to be compared is relative to the
"InterfaceData" dictionary.

Each operator, when given a filter argument, takes an array
(dictionary containing only one type of item) as its first argument.
In the current example, this would be "Interfaces". The items in it
are all of type "InterfaceData". This tag is referred to as the
"iteration tag".

Before a filtered operation is used, BEGIN must be used to put the
array (dictionary) on top of the stack, to establish it as the
context that the filter iterates over. The general operation of a
filtered operation is then:

     1. Iterate over the items in the array.

     2. For each element in the array, execute the filter.

     3. If the filter succeeds, do the requested operation
        (GET/SET/etc.) on the matched element, using the
        template/value/path as input to the operation. At this
        point, the execution of the operation is the same as in
        the non-filtered case.

This is a model of operation; actual implementations may take
advantage of whatever lookup techniques are available for the
particular table (array) involved.

Therefore, there are three inputs to a filtered operation:

     1. The "current dictionary" on the stack. This is the
        array-type dictionary to be searched, set by an earlier
        BEGIN.

     2. A filter, to test each item in the array. Each path or
        value mentioned in the filter must be named in the context

of an item in the array, as if it was the current
dictionary. For example, in the case where a filtered
operation iterates over the set of "InterfaceData" items
in the "Interfaces" array, each value or path in the
filter should name an item in the "InterfaceData"
dictionary, such as "address".

   3. A template, path, or value associated with the operation
      to be performed. The leading ASN.1 tag in this must match
      the iteration tag. In the current example where the
      filter is searching the "Interfaces" dictionary, the first
      tag in the template/tag/value must be "InterfaceData".

The operators which take filters as arguments are:

BEGIN   array path filter   BEGIN   array dict
         Find a dictionary in <array> that matches <filter>. Use
         that as the starting point for <path> and push the
         dictionary corresponding to <path> onto the stack. If more
         than one dictionary matches <filter>, then any of the
         matches may be used. This specification does not state how
         the choice is made. At least one dictionary must match; it
         is an error if there are no matches. (Perhaps it should be
         an error for there to be multiple matches; actual
         experience is needed to decide.)

GET   array template filter   GET   array
         For each item in <array> that matches <filter>, fill in the
         template with values from the data tree and emit the
         result. The first tag of <template> must be equal to the
         iteration tag. Selected parts of matched items are emitted
         based upon <template>, just as in a non-filtered GET
         operation.

GET-ATTRIBUTES
           array template filter   GET-ATTRIBUTES   array
         Same as GET, except emit attributes rather than data
         values.

SET   array value filter   SET   array
         Same as GET, except set the values in <value> rather than
         retrieving values. Several values in the data tree will be
         changed if the filter matches more than one item in the
         array.

DELETE   array filter   DELETE   array
         Delete the entry(s) in <array> that match <filter>.

Notes about filter execution:

- Expressions are executed by inorder tree traversal.

- Since the filter operations are all GETs and comparisons,
  there are no side-effects to filter execution, so an
  implementation is free to execute only as much of the
  filter as required to produce a result (e.g., don't execute
  the rest of an AND if the first comparison turns out to be
  false).

- It is not an error for a filter to test a data item that
  isn't in the data tree. In this situation, the comparison
  just fails (is false). This means that filters don't need
  to test for the existence of optional data before
  attempting to compare it.

Here is an example of how filtering would be used to obtain the input
and output packet counts for the interface with IP address 10.0.0.51.

```
Interfaces BEGIN                      -- dictionary
InterfaceData{ pktsIn, pktsOut }      -- template
Filter{ equal{ address(10.0.0.51) } }
GET
END                                   -- finished with dict
```

The returned value would be something like:

```
Interfaces{                           -- BEGIN
  InterfaceData{ pktsIn(1345134), pktsOut(1023729) }
                                      -- GET
}                                     -- END
```

The annotations indicate which part of the response is generated by
the different operators in the query.

Here is an example of accessing a table contained within some other
table. Suppose we want to get at the ARP table for the interface
with IP address 36.8.0.1 and retrieve the entire ARP entry for the
host with IP address 36.8.0.23. In order to retrieve a single entry
in the ARP table (using a filtered GET), a BEGIN must be used to get
down to the ARP table. Since the ARP table is contained within the
Interfaces dictionary (an array), a filtered BEGIN must be used.

```
Interfaces BEGIN                      -- dictionary
InterfaceData{ ARP }                  -- path
Filter{ equal{ address(36.8.0.1) } } -- filter
BEGIN                                 -- filtered BEGIN
```

```
     -- Now in ARP table for 38.0.0.1; get entry for 38.8.0.23.
     addrMap                               -- whole entry
     Filter{ equal{ ipAddr(36.8.0.23) } }    -- filter
     GET                                   -- filtered GET
     END
     END
```

   The result would be:

```
     Interfaces{                           -- first BEGIN
       InterfaceData{ ARP{                 -- second BEGIN
         addrMap{ ipAddr(36.8.0.23), physAddr(..) } -- from GET
       } }                                 -- first END
     }                                     -- second END
```

   Note which parts of the output are generated by different parts of
   the query.

   Here is an example of how the SET operator would be used to shut down
   the interface with ip-address 10.0.0.51 by changing its status to
   "down".

```
     Interfaces BEGIN                      -- get dictionary
     Interface{ Status(down) }             -- value to set
     Filter{ equal{ IP-addr(10.0.0.51) } } }
     SET
     END
```

   If the SET is successful, the result would be:

```
     Interfaces{                           -- BEGIN
         Interface{ Status(down) }         -- from SET
     }                                     -- END
```

8.7 Terminating a Query

   A query is implicitly terminated when there are no more ASN.1 objects
   to be processed by the interpreter. For a perfectly-formed query,
   the interpreter would be back in the state it was when it started:
   the stack would have only the root dictionary on it, and all of the
   ASN.1 objects in the result would be terminated.

   If there are still "open" ASN.1 objects in the result (caused by
   leaving ENDs off of the query), then these are closed, as if a
   sufficient number of ENDs were provided. This condition would be
   indicated by the existence of dictionaries other than the root
   dictionary on the stack.

If an extra END is received that would pop the root dictionary off of
the stack, the query is terminated immediately. No error is
generated.

9. EXTENDING THE SET OF VALUES

There are two ways to extend the set of values understood by the
query language. The first is to register the data and its meaning
and get an ASN.1 tag assigned for it. This is the preferred method
because it makes that data specification available for everyone to
use.

The second method is to use the VendorSpecific application type to
"wrap" the vendor-specific data. Wherever an implementation defines
data that is not in RFC-1024, the "VendorSpecific" tag should be used
to label a dictionary containing the vendor-specific data. For
example, if a vendor had some data associated with interfaces that
was too strange to get standard numbers assigned for, they could,
instead represent the data like this:

```
        interfaces {
                interface {
                        in-pkts, out-pkts, ...
                        VendorSpecific { ephemeris, declination }
                        }
                }
```

In this case, ephemeris and declination correspond to two context-
dependent tags assigned by the vendor for their non-standard data.

If the vendor-specific method is chosen, the private data MUST have
descriptions available through the GET-ATTRIBUTES operator. Even
with this descriptive ability, the preferred method is to get
standard numbers assigned if possible.

10. AUTHORIZATION

This specification does not state what type of authorization system
is used, if any. Different systems may have needs for different
mechanisms (authorization levels, capability sets, etc.), and some
systems may not care about authorization at all. The only effect
that an authorization system has on a query is to restrict what data
items in the tree may be retrieved or modified.

Therefore, there are no explicit query language features that deal
with protection. Instead, protection mechanisms are implicit and may
make some of the data invisible (for GET) or non-writable (for SET):

- Each query runs with some level of authorization or set of
  capabilities, determined by its environment (HEMS and the
  HEMP header).

- Associated with each data item in the data tree is some
  sort of test to determine if a query's authorization should
  grant it access to the item.

Authorization tests are only applied to query language operations
that retrieve information (GET, GET-ATTRIBUTES, and GET-RANGE) or
modify it (SET, CREATE, DELETE). An authorization system must not
affect the operation of BEGIN and END. In particular, the
authorization must not hide entire dictionaries, because that would
make a BEGIN on such a dictionary fail, terminating the entire query.

11. ERRORS

If some particular information is requested but is not available, it
will be returned as "no-value" by giving the ASN.1 length as 0.

When there is any other kind of error, such as having improper
arguments on the top of the stack or trying to execute BEGIN when the
path doesn't refer to a dictionary, an ERROR object is emitted.

The contents of this object identify the exact nature of the error:

```
        Error ::= [APPLICATION 0] IMPLICIT SEQUENCE {
                errorCode       INTEGER,
                errorInstance   INTEGER,
                errorOffset     INTEGER
                errorDescription IA5String,
                errorOp         INTEGER,
                }
```

errorCode identifies what the error was, and errorInstance is an
implementation-dependent code that gives a more precise indication of
where the error occured. errorOffset is the location within the
query where the error occurred. If an operation was being executed,
errorOp contains its operation code, otherwise zero.
errorDescription is a text string that can be printed that gives some
description of the error. It will at least describe the errorCode,
but may also give details implied by errorInstance. Detailed
definitions of all of the fields are given in appendix I.2.

Since there may be several unterminated ASN.1 objects in progress at
the time the error occurs, each one must be terminated. Each
unterminated object will be closed with a copy of the ERROR object.
Depending upon the type of length encoding used for this object, this

will involve filling the value for the length (definite length form)
or emitting two zero octets (indefinite length form). After all
objects are terminated, a final copy of the ERROR object will be
emitted. This structure guarantees that the error will be noticed at
every level of interpretation on the receiving end.

In summary, if there was an error before any ASN.1 objects were
generated, then the result would simply be:
      error{...}

If a couple of ASN.1 objects were unterminated when the error
occurred, the result might look like:
      interfaces{
            interface { name(...) type(...) error{...} }
            error{...}
            }
      error{...}

It would be possible to define a "WARNING" object that has a similar
(or same) format as ERROR, but that would be used to annotate
responses when a non-fatal "error" occurs, such as attempting to
SET/CREATE/DELETE and the operation is denied. This would be an
additional complication, and we left it out in the interests of
simplicity.

I. ASN.1 DESCRIPTIONS OF QUERY LANGUAGE COMPONENTS

A query consists of a sequence of ASN.1 objects, as follows:

      Query := IMPLICIT SEQUENCE of QueryElement;

      QueryElement ::= CHOICE {
            Operation,
            Filter,
            Template,
            Path,
            InputValue
            }

Operation and Filter are defined below. The others are:

      Template   ::= any
      Path   ::= any
      InputValue   ::= any

These three are all similar, but have different restrictions on their
structure:

    Template   Specifies a portion of the tree, naming one or more
                      values, but not containing any values.

    Path   Specifies a single path from one point in the tree to
                      another, naming exactly one value, but not containing
                      a value.

    InputValue   Gives a value to be used by a query language
                      operator.

    A query response consists of a sequence of ASN.1 objects, as follows:

        Response := IMPLICIT SEQUENCE of ResponseElement;

        ResponseElement ::= CHOICE {
                ResultValue,
                Error
                }

    Error is defined below. The others are:

        ResultValue   ::= any

    ResultValue is similar to Template, above:

    ResultValue   Specifies a portion of the tree, naming and
                      containing one or more values.

    The distinctions between these are elaborated in section 6.

I.1 Operation Codes

    Operation codes are all encoded in a single application-specific
    type, whose value determines the operation to be performed. The
    definition is:

        Operation ::= [APPLICATION 1] IMPLICIT INTEGER {
                reserved(0),
                begin(1),
                end(2),
                get(3),
                get-attributes(4),
                get-range(5),
                set(6),

```
            create(7),
            delete(8)
            }
```

I.2 Error Returns

   An Error object is returned within a reply when an error is
   encountered during the processing of a query. Note that the
   definition this object is similar to that of the HEMP protocol error
   structure. The error codes have been selected to keep the code
   spaces distinct between the two. This is intended to ease the
   processing of error messages. See section 11 for more information.

```
      Error ::= [APPLICATION 0] IMPLICIT SEQUENCE {
              errorCode        INTEGER,
              errorInstance    INTEGER,
              errorOffset      INTEGER
              errorDescription IA5String,
              errorOp          INTEGER,
              }
```

   The fields are defined as follows:

   errorCode   Identifies the general cause of the error.

   errorInstance   An implementation-dependent code that gives a more
                     precise indication of where the error occured in the
                     query processor. This is most useful when internal
                     errors are reported.

   errorOffset   The location within the query where the error was
                     detected. The first octet of the query is numbered
                     zero.

   errorOp   If an operation was being executed, this contains its
                     operation code, otherwise zero.

   errorDescription
                     A text string that can be printed that gives some
                     description of the error. It will at least describe
                     the errorCode, but may also give details implied by
                     errorInstance.

   Some errors are associated with the execution of specific operations,
   and others with the overall operation of the query interpreter. The
   errorCodes are split into two groups.

   The first group deals with overall interpreter operation. Except for

"unknown operation", these do not set errorOp.

100   Other error.
                 Any error not listed below.

101   Format error.
                 An error has been detected in the format of the input
                 stream, preventing further interpretation of the
                 query.

102   System error.
                 The query processor has failed in some way due to an
                 internal error.

103   Stack overflow.
                 Too many items were pushed on the stack.

104   Unknown operation.
                 The operation code is invalid. errorOp is set.

The second group is errors that are associated with the execution of
particular operations. errorOp will always be set for these.

200   Other operation error.
                 Any error, associated with an operation, not listed
                 below.

201   Stack underflow.
                 An operation expected to see some number of operands
                 on the stack, and there were fewer items on the
                 stack.

202   Operand error.
                 An operation expected to see certain operand types on
                 the stack, and something else was there.

203   Invalid path for BEGIN.
                 A path given for BEGIN was invalid, because some
                 element in the path didn't exist.

204   Non-dictionary for BEGIN.
                 A path given for BEGIN was invalid, because the given
                 node was a leaf node, not a dictionary.

205   BEGIN on array element.
                 The path specified an array element. The path must
                 point at a single, unique, node. A filtered BEGIN
                 should have been used.

      206   Empty filter for BEGIN.
                       The filter for a BEGIN didn't match any array
                       element.

      207   Filtered operation on non-array.
                       A filtered operation was attempted on a regular
                       dictionary. Filters can only be used on arrays.

      208   Index out of bounds.
                       The starting address or length for a GET-RANGE
                       operation went outside the bounds for the given
                       object.

      209   Bad object for GET-RANGE.
                       GET-RANGE can only be applied to objects whose base
                       type is OctetString.

   This list is probably not quite complete, and would need to be
   extended, based upon implementation experience.

I.3 Filters

   Many of the operations can take a filter argument to select among
   elements in an array. They are discussed in section 8.6.

      Filter   ::= [APPLICATION 2] CHOICE {
                              present   [0] DataPath,
                              equal   [1] DataValue,
                              greaterOrEqual [2] DataValue,
                              lessOrEqual   [3] DataValue,
                              and   [4] SEQUENCE OF Filter,
                              or   [5] SEQUENCE OF Filter,
                              not   [6] Filter
                              }

      DataPath   ::= ANY   -- Path with no value

      DataValue   ::= ANY   -- Single data value

   A filter is executed by inorder traversal of its ASN.1 structure.

   The basic filter operations are:

   present   tests for the existence of a particular data item in
                   the data tree

    equal   tests to see if the named data item is equal to the
                        given value.

    greaterOrEqual   tests to see if the named data item is greater than
                        or equal to the given value.

    lessOrEqual   tests to see if the named data item is less than or
                        equal to the given value.

    These may be combined with "and", "or", and "not" operators to form
    arbitrary boolean expressions. The "and" and "or" operators will
    take any number of terms. Terms are only evaluated up to the point
    where the outcome of the expression is determined (i.e., an "and"
    term's value is false or an "or" term's value is true).

I.4 Attributes

    One or more Attributes structure is returned by the GET-ATTRIBUTES
    operator. This structure provides descriptive information about
    items in the data tree. See the discussion in section 8.3.

        Attributes ::= [APPLICATION 3] IMPLICIT SEQUENCE {
                tagASN1   [0] IMPLICIT INTEGER,
                valueFormat   [1] IMPLICIT INTEGER,
                longDesc   [2] IMPLICIT IA5String OPTIONAL,
                shortDesc   [3] IMPLICIT IA5String OPTIONAL,
                unitsDesc   [4] IMPLICIT IA5String OPTIONAL,
                precision   [5] IMPLICIT INTEGER OPTIONAL,
                properties   [6] IMPLICIT BITSTRING OPTIONAL,
                valueSet   [7] IMPLICIT SET OF valueDesc OPTIONAL
                }
        valueDesc ::= IMPLICIT SEQUENCE {
                value   [0] ANY,   -- Single data value
                desc   [1] IA5String
                }

    The meanings of the various attributes are given below.

    tagASN1   The ASN.1 tag for this object. This attribute is
                    required.

    valueFormat   The underlying ASN.1 type of the object (e.g.,
                    SEQUENCE or OCTETSTRING or Counter). This is not
                    just the tag number, but the entire tag, as it would
                    appear in an ASN.1 object. As such, it includes the
                    class, which should be either UNIVERSAL or
                    APPLICATION. Applications receiving this should

                    ignore the constructor bit. This attribute is
                    required.

   longDesc   A potentially lengthy text description which fully
                    defines the object. This attribute is optional for
                    objects defined in this memo and required for
                    entity-specific objects.

   shortDesc   A short mnemonic string of less than 15 characters,
                    suitable for labeling the value on a display. This
                    attribute is optional.

   unitsDesc   A short string used for integer values to indicate
                    the units in which the value is measured (e.g., "ms",
                    "sec", "pkts", etc.). This attribute is optional.

   precision   For Counter objects, the value at which the Counter
                    will roll-over. Required for all Counter objects.

   properties   A bitstring of boolean properties of the object. If
                    the bit is on, it has the given property. This
                    attribute is optional. The bits currently defined
                    are:

                    0   If true, the difference between two values of
                         this object is significant. For example, the
                         changes of a packet count is always significant,
                         it always conveys information. In this case, the
                         0 bit would be set. On the other hand, the
                         difference between two readings of a queue length
                         may be meaningless.

                    1   If true, the value may be modified with SET,
                         CREATE, and DELETE. Applicability of CREATE and
                         DELETE depends upon whether the object is in an
                         array.

                    2   If true, the object is a dictionary, and a BEGIN
                         may be used on it. If false, the object is leaf
                         node in the data tree.

                    3   If true, the object is an array-type dictionary,
                         and filters may be used to traverse it. (Bit 2
                         will be true also.)

   valueSet   For data that is defined as an ASN.1 CHOICE type (an
                    enumerated type), this gives descriptions for each of
                    the possible values that the data object may assume.

                         Each valueDesc is a <value,description> pair. This
                         information is especially important for control
                         items, which are very likely to appear in
                         VendorSpecific dictionaries, exactly the situation
                         where descriptive information is needed.

I.5 VendorSpecific

   See the discussion in section 9.

        VendorSpecific   ::= [APPLICATION 4] IMPLICIT SET
                                      of ANY

II. IMPLEMENTATION HINTS

   Although it is not normally in the spirit of RFCs to define an
   implementation, the authors feel that some suggestions will be useful
   to implementors of the query language. This list is not meant to be
   complete, but merely to give some hints about how the authors imagine
   that the query processor might be implemented efficiently.

      - It should be understood that the stack is of very limited
        depth. Because of the nature of the query language, it can
        get only about 4 entries (for arguments) plus the depth of
        the tree (up to one BEGIN per level in the tree). This
        comes out to about a dozen entries in the stack, a modest
        requirement.

      - The stack is an abstraction -- it should be implemented
        with pointers, not by copying dictionaries, etc.

      - An object-oriented approach should make implementation
        fairly easy. Changes to the "shape" if the data items
        (which will certainly occur, early on) will also be easier
        to make.

      - Only a few "messages" need to be understood by objects. By
        having pointers to action routines for each basic operation
        (GET,SET,...) associated with each node in the tree, common
        routines (e.g., emit a long integer located at address X)
        can be shared, and special routines (e.g., set the interface
        state for interface X) can be implemented in a common
        framework. Higher levels need know nothing about what data
        is being dealt with.

      - Most interesting objects are dictionaries, each of which
        can be implemented using pointers to the data and procedure
        "hooks" to perform specific operations such as GET, SET,

          filtering, etc.

      - The hardest part is actually extracting the data from
        existing TCP/IP implementations that weren't designed with
        detailed monitoring in mind. Query processors interfacing
        to a UNIX kernel will have to make many system calls in
        order to extract some of the more intricate structures,
        such as routing tables. This should be less of a problem
        if a system is designed with easy monitoring as a goal.

A Skeletal Implementation

   This section gives a rather detailed example of the core of a query
   processor. This code has not been tested, and is intended only to
   give implementors ideas about how to tackle some aspects of query
   processor implementation with finesse, rather than brute force.

   The suggested architecture is for each dictionary to have a
   "traverse" routine associated with it, which is called when any sort
   of operation has to be done on that dictionary. Most nodes will
   share the same traversal routine, but array dictionaries will usually
   have routines that know about whatever special lookup mechanisms are
   required.

   Non-dictionary nodes would have two routines, "action", and
   "compare", which implement query language operations and filter
   comparisons, respectively. Most nodes would share these routines.

   For example, there should be one "action" routine that does query
   language operations on 32-bit integers, and another that works on
   16-bit integers, etc.

   Any traversal procedure would take arguments like:

        traverse(node, mask, op, filter)
                Treenode   node;   /* generic node-in-tree */
                ASN   mask;   /* internal ASN.1 form */
                enum opset   op;   /* what to do */
                Filter   filter; /* zero if no filter */

        enum opset { begin, get, set, create, delete, geta,
                     c_le, c_ge, c_eq, c_exist };

   The traversal procedure is called whenever anything must be done
   within a dictionary. The arguments are:

   node   the current dictionary.

mask   is either the template, path, or value, depending
                   upon the operation being performed. The top-level
                   identifier of this object will be looked up in the
                   context of <node>.

op   is the operation to be performed, either one of the
                   basic operations, or a filter operation.

filter   is the filter to be applied, or zero if none. There
                   will be no filter when <op> is a filter-comparison
                   operation.

The general idea is that the traversal proc associated with a node
has all of the knowledge about how to get around in this subtree
encoded within it. Hopefully, this will be the only place this
knowledge is coded. Here is a skeleton of the "standard" traversal
proc, written mostly in C.

When the query processor needs to execute a "GET" operation, it would
just call:
     traverse(current, template, GET, 0)

Notes about this example:

     - This traversal routine handles either query language
       operations (GET, SET, etc.) or low-level filter operations.
       Separate routines could be defined for the two classes of
       operations, but they do much of the same work.

     - Dictionary nodes have a <traversal> proc defined.

     - Leaf nodes have an <action> proc, which implement GET, SET,
       GET-ATTRIBUTES, CREATE, and DELETE, and a <compare> proc,
       which performs low-level filter comparisons.

     - In the generic routine, the filter argument is unused,
       because the generic routine isn't used for array
       dictionaries, and only array dictionaries use filters.

     - An ASN type contains the top level tag and a list of
       sub-components.

     - size(mask) takes an ASN.1 object and tells how many
       sub-items are in it. Zero means that this is a simple
       object.

     - lookup(node, tag) looks up a tag in the given (tree)node,
       returning a pointer to the node. If the tag doesn't exist

Trewitt & Partridge   [Page 38]

        in that node, a pointer to a special node "NullItem" is
        returned. NullItem looks like a leaf node and has procs
        that perform the correct action for non-existent data.

      - This example does not do proper error handling, or ASN.1
        generation, both of which would require additional code in
        this routine.

```
 /*
  *  For op = GET/SET/etc, return:
  *              true on error, otherwise false.
  *  When op is a filter operation, return:
  *              the result of the comparison.
  */
 int std_traverse(node, mask, op, filter)
     Treenode    node;    /* current node */
     ASN         mask;    /* internal ASN.1 form */
     enum opset  op;      /* what to do */
     Filter      filter;  /* unused in this routine */
 {
     ASN         item;
     Treenode    target;
     boolean     rv = false;
     extern Treenode NullItem;

     if (filter != null) {
         error(...);
         return true;
     }

     target = lookup(node, mask.tag);

     /*  We are at the leaf of the template/path/value.  */
     if (size(mask) == 0)
         switch (op)
         {
         case BEGIN:
             /*  non-existent node, or leaf node  */
             if (target == NullItem || target.traverse == 0) {
                 error(...);
                 return true;
                 }
             else {
                 begin(node, mask.tag);
                 return false;
                 }

         case GET:      case SET:      case GETA:
```

```
            case GETR:      case CREATE:    case DELETE:
                /*  A leaf in the mask specifies entire directory.
                    For GET, traverse the entire subtree.  */
                if (target.traverse)
                    if (op == GET) {
                        foreach subnode in target
                            /*  Need to test to not GET memory.  */
                            rv |= (*target.traverse)
                                      (target, subnode.tag, op, 0);
                        return rv;
                    }
                    else if (op == SET)     /*  no-op  */
                        return false;
                    else if (op != GETA) {
                        error(...);
                        return true;
                    }
                /*  We're at a leaf in both the mask and the tree.
                    Just execute the operation.
                */
                else {
                    if (op == BEGIN) {  /*  Can't begin on leaf  */
                        error(...);
                        return true;
                    else
                        return (*target.action)(target, mask, op);
                    }
                }  /* else */

        default:          /*  Comparison ops.  */
            return (*target.compare)(target, mask, op);
        }  /* switch */

    /*  We only get here if mask has structure.  */

    /*  can't have multiple targets for BEGIN  */
    if (op == BEGIN && size(mask) != 1) {
        error(...);
        return true;
    }
    /*  or for a single filter operation.  */
    if (op is comparison && size(mask) != 1) {
        error(...);
        return false;
    }
    /*  Iterate over the components in mask  */
    foreach item in mask
    {
```

```
            if (target.traverse)    /*  traverse subtree.  */
                rv |= (*component.traverse)(component, item, op, 0);
            else                      /*  leaf node, at last.  */
                if (op is comparison)
                    return (*target.compare)(target, mask, op);
                else
                    return (*target.action)(target, mask, op);
        } /* foreach */

        return rv;
    }  /* std_traverse */
```

   Here is a bare skeleton of an array-type dictionary's traversal proc.

```
    int array_traverse(node, mask, op, filter)
        Treenode    node;   /* current node */
        ASN         mask;   /* internal ASN.1 form */
        enum opset  op;     /* what to do */
        Filter      filter; /* unused in this routine */
    {
        Treenode    target;
        boolean     rv = false;
        extern Treenode NullItem;

        /*  Didn't find that key.  */
        if (mask.tag != this array's iteration tag)
            return false;

        if (op == BEGIN && filter == null) {
            error(...);
            return 1;
        }

        /*  The implementation of this loop is the major trick!  */
        /*  Needs to stop after first filter success on BEGIN.  */
        foreach target in node {
            if (filter == null ||           /*  if no filter, or */
                ExecFilter(target, filter)) /* if it succeeds  */
                rv |= (target.traverse*)(target, mask, op, 0);
        }
    }  /* array_traverse */
```

   Object-oriented programming languages, such as C++, Modula, and Ada,
   are well suited to this style of implementation. There should be no
   particular difficulty with using a conventional language such as C or
   Pascal, however.

III. OBTAINING A COPY OF THE ASN.1 SPECIFICATION

    Copies of ISO Standard ASN.1 (Abstract Syntax Notation 1) are
    available from the following source. It comes in two parts; both are
    needed:

        IS 8824 -- Specification (meaning, notation)
        IS 8825 -- Encoding Rules (representation)

    They are available from:

        Omnicom Inc.
        115 Park St, S.E.   (new address as of March, 1987)
        Vienna, VA 22180
        (703) 281-1135