Network Working Group
Request for Comments: 1023

G. Trewitt
Stanford
C. Partridge
BBN/NNSC
October 1987

HEMS Monitoring and Control Language

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.

STATUS OF THIS MEMO

The purpose of this RFC is provide a specification for monitoring and control of network entities in the Internet. This is an experimental specification and is intended for use in testing the ideas presented here. No proposals in this memo are intended as standards for the Internet at this time. After sufficient experimentation and discussion, this RFC will be redrafted, perhaps as a standard. Distribution of this memo is unlimited.

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

OVERVIEW AND SCOPE

The basic model of monitoring and control used in this proposal 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 things, or both. We will use gateway-oriented examples, but it should be understood that this query-response mechanism can be applied to other entities besides just gateways.

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.

It is not necessary for a monitored entity to be able to store the complete query. It is quite possible for an implementation to process the query on the fly, producing portions of the response while the query is still being received.

Other RFCs associated with HEMS are: RFC-1021 -- Overview; RFC-1022 -- transport protocol and message encapsulation; RFC-1024 -- precise data definitions. These issues are not dealt with here. 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.

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 of this notation, see the ISO documents IS 8824 and IS 8825. See the end of this memo for information about ordering these documents.

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. The most important difference is that, in this memo, we are not concerned with the length of the data items.

ASN.1 data items may be either a "simple type" such as integer or octet string or a "structured type", a collection of data items. The notation or a "structured type", a collection of data items. The notation:

```
     ID(value)
```

represents a simple data item whose tag is "ID" with the given value. A structured data item is represented as:

```
     ID { ... contents ... }
```

where contents is a sequence of data items. Remember that the contents may include both simple and structured types, so the structure is fully recursive.

There are situations where it is desirable to specify a type but give no value, such as when there is no meaningful value for a particular measured parameter or when the entire contents of a structured type is being specified. In this situation, the same notation is used,

but with the value omitted:

```
ID()
```

or

```
ID{}
```

The representation of this is obvious -- the data item has zero for the length and no contents.

DATA MODEL

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 "constructor" types as an envelope for an entire subtree.

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

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 would just be an ASN.1 tag. Such names would be small integers (typically, less than 100) and so could easily be mapped by the monitored entity onto its internal representation.

We will use the term "dictionary" to represent an internal node in the hierarchy. Here is a possible organization of the hierarchy in an entity that has several network interfaces and multiple processes. The exact organization of data in entities is specified in RFC-1024.

```
        system {
                name                            -- host name
                clock-msec                      -- msec since boot
                interfaces                      -- # of interfaces
                }
        interfaces {                    -- one per interface
                interface { type, ip-addr, in-pkts, out-pkts, . . . }
                interface { type, ip-addr, in-pkts, out-pkts, . . . }
                interface { type, ip-addr, in-pkts, out-pkts, . . . }
                                :
                }
        processes {
                process { name, stack, interrupts, . . . }
                process { name, stack, interrupts, . . . }
                                :
                }
        route-table {
                route-entry { dest, interface, nexthop, cost, . . . }
                route-entry { dest, interface, nexthop, cost, . . . }
                                :
                }
        arp-table {
                arp-entry { hard-addr, ip-addr, age }
                arp-entry { hard-addr, ip-addr, age }
                                :
                }
        memory { }
```

The "name" of the clock in this entity would be:

```
        system{ clock-msec }
```

and the name of a route-entry's IP address would be:

```
        route-table{ route-entry{ ip-addr } }.
```

Actually, this is the name of the IP addresses of ALL of the routing table entries. This ambiguity is a problem in any situation where there are several instances of an item being monitored. If there was a meaningful index for such tabular data (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. More on this later.

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:

```
        route-table{ route-entry{ ip-addr, cost } }.
```

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, processes, routing tables, and ARP tables are all arrays. In fact, we expect that most of the interesting data in an entity will be contained in arrays.

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. Arrays are usually accessed associatively using special operators in the language.

The fact that these objects are viewed externally as arrays 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 represented as, for example, a hash table.

REPRESENTATION OF A REPLY

The data returned to the monitoring entity is a sequence of ASN.1 data items. Each of these corresponds to one the top-level dictionaries maintained by the monitored entity. The tags for these data items will be in the "application-specific" class (e.g., if an entity has the above structure for its data, then the only top-level data items that will be returned will have tags corresponding to these groups). If a query returned data from two of these, the representation might look like:

```
       interfaces{ . . . }  route-table{ . . . }
```

which is just a stream of two ASN.1 objects (each of which may consist of many sub-objects).

Data not in the root dictionary will have tags from the context-specific class. Therefore, data must always be fully qualified. For example, the name of the entity would 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 were "name" inside the "system" dictionary or "dest" inside the "interfaces" dictionary (assuming in this case that "name" and "dest" were assigned the same ASN.1 tag).

Having fully-qualified data simplifies decoding of the data at the receiving end and allows the tags to be locally chosen (e.g., definitions for tags dealing with ARP tables can't conflict with definitions for tags dealing with interfaces). Therefore, the people

doing the name assignments are less constrained. In addition, most of the identifiers will be fairly small integers.

It will often be the case that requested data may not be available, either because the request was badly formed (asked for data that couldn't exist) or because the particular data item wasn't defined in a particular situation (time since last error, when there hasn't been an error). In this situation, the returned data item will have the same tag as in the request, but will have zero-length data. Therefore, there can NEVER be an "undefined data" error.

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

REPRESENTATION OF A REQUEST

A request to a monitored entity is also a sequence of ASN.1 data items. Each item will fit into one of the following categories:

Template	These are objects with the same types as the objects returned by a request. The difference is that a template only specifies the shape of the data -- there are no values contained in it. Templates are used to select specific data to be returned. No ordering of returned data is implied by the ordering in a template. A template may be either simple or structured, depending upon what data it is naming. The representations of the simple data items in a template all have a length of zero.
Tag	A tag is a special case of a template that is a simple (non-structured) type (i.e., it names exactly one node in the dictionary tree).
Opcodes	These objects tell the query interpreter to do something. They are described in detail later in this report. Opcodes are represented as an application-specific type whose value determines the operation. These values are defined in RFC-1024.
Data	These are the same objects that are used to represent information returned from an entity. It is occasionally be necessary to send data as

part of a request. For example, when requesting information about the interface with IP address "10.0.0.51", the address would be sent in the same format in the request as it would be seen in a reply.

Data, Tags, and Templates are usually in either the context-specific class, except for items in the root dictionary and a few special cases, which are in the application-specific class.

QUERY LANGUAGE

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.

In the current version of the query language:

- There are no subroutines.
- There are no control structures defined in the language.
- There are no arithmetic or conditional operators.

These features could be added to the language if needed.

This language is designed with the goal of being expressive enough to write useful queries with, but to guarantee simplicity, both of query execution and language implementation.

The central element of the language is the stack. It may contain templates, (and therefore tags), data, or dictionaries (and therefore arrays) from the entity being monitored. Initially, 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 things from the stack and may generate ASN.1 objects and send them to the output stream. Note that portions of the response may be generated while the query is still being received.

The following opcodes are defined in the language. This is a

provisional list -- changes may need to be made to deal with additional needs.

In the descriptions below, opcode 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 t
means that the OP operator takes <a> and <b> off of the stack and leaves <t> on the stack. Many of the operators below leave the first operand (<a> in this example) on the stack for future use.

Here are the operators defined in the query language:

GET dict template GET dict
Emit an ASN.1 object with the same "shape" as the given template. Any items in the template that are not in <dictionary> (or its components) are represented as objects with a length of zero. This handles requests for data that isn't available, either because it isn't defined or because it doesn't apply in this situation.

or 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.

BEGIN dict1 tag BEGIN dict1 dict
Pushes the value for dict{ tag } on the stack, which should be another dictionary. At the same time, produce the beginning octets of an ASN.1 object corresponding to that dictionary. It is up to the implementation to choose between using the "indefinite length" representation or going back and filling the length in later.

END dict END --
Pop the dictionary off of the stack and terminate the currently open ASN.1 object. Must be paired with a BEGIN.

Getting Items Based on Their Values

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 {
                interface { type, ip-addr, in-pkts, out-pkts, ...}
                interface { type, ip-addr, in-pkts, out-pkts, ...}
                                :
                                :
                }
```

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 is to just number the interfaces and refer to the desired interface as:

```
        interfaces(3)
```

for the third one.

But this is probably not sufficient since interface numbers may change over time, perhaps from one reboot to the next. This method is not sufficient at all for arrays with many elements, such as processes, routing tables, 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 in the language for indexing.

A better scheme is to select objects based upon some value contained in them, such as the IP address or process name. The GET-MATCH operator provides this functionality in a fairly general way.

GET-MATCH array value template GET-MATCH array
: <array> should be a array (dictionary containing only one type of item). The first tag in <value> and <template> must match this type. For each entry in <array>, match the <value> against the contents of the entry. If there is a match, emit the entry based upon <template>, just as in a GET operation.

If there are several leaf items in the value to be matched against, as in:

```
        route-entry{ interface(1), cost(3) }
```

all of them must match an array entry for it to be emitted.

Here is an example of how this operator would be used to obtain the input and output packet counts for the interface with ip-address 10.0.0.51.

```
interfaces BEGIN                        -- get dictionary
interface{ ip-addr(10.0.0.51) }         -- value to match
interface{ in-pkts out-pkts }           -- data template to get
GET-MATCH
END                                     -- finished with dict
```

The exact meaning of a "match" is dependent upon the characteristics of the entities being compared. In almost all cases, it is a comparison for exact equality. However, it is quite reasonable to define values that allow matches to do interesting things. For example, one might define three different flavors of "ip-addr": one that has only the IP net number, one with the IP net+subnet, and the whole IP address. Another possibility is to allow for wildcards in IP addresses (e.g., if the "host" part of an IP address was all ones, then that would match against any IP address with the same net number).

So, for all data items defined, the behavior of the match operation must be defined if it is not simple equality.

Implementations don't have to provide the ability to use all items in an object to match against. It is expected that some data structures that provide for efficient lookup for one item may be very inefficient for matching against others. (For instance, routing tables are designed for lookup with IP addresses. It may be very difficult to search the routing table, matching against costs.)

NOTE: It would be desirable to provide a general-purpose filtering capability, rather than just "equality" as provided by GET-MATCH. However, because of the potential complexity of such a facility, lack of a widely-accepted representation for filter expressions, and time pressure, we are not defining this mechanism now.

However, if a generalized filtering mechanism is devised, the GET-MATCH operator will disappear.

Data Attributes

Although ASN.1 data is self-describing as far as the structure goes, it gives no information about what the data means (e.g., By looking at the raw data, it is possible to tell that an item is of type [context 5] and 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?).

Most of the time, this information will come from RFC-1024, which defines all of the ASN.1 tags and their precise meaning. When extensions have been made, it may not be possible to get

documentation on the extensions. (See the section about extensions, page 15.)

The query language provides a set of operators parallel to the GET and GET-MATCH operators that return a set of attributes describing the data. This information should be sufficient to let a human understand the meaning of the data and to let a sophisticated application treat the data appropriately. The information is sufficient to let an application format the information on a display and decide whether or not 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 the 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 just look exactly like the data itself. The format is:

```
Attributes ::= [APPLICATION 2] 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,
}
```

For example, the attributes for

```
    system{ name, clock-msec }
```

might be:

```
    system{
            Attributes{
                    tagASN1(name), valueFormat(IA5String),
                    longDesc("The name of the host"),
                    shortDesc("hostname")
            },
            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. A complete definition of the contents of the Attributes type is in RFC-1024.

Note that there will 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.

GET-ATTRIBUTES
dict template GET-ATTRIBUTES dict
Emit ASN.1 Attributes objects that for the objects named in <template>. Any items in the template that are not in <dictionary> (or its components), elicit an Attributes object with no.

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.

GET-ATTRIBUTES-MATCH
dict value template GET-ATTRIBUTES-MATCH dict <array>
should be an array (dictionary containing only one type of item). The first tag in <value> and <template> must match this type. For each entry in <array>, match the <value> against the contents of the entry. If there is a match, emit the atributes based upon <template>, just as in a GET-ATTRIBUTES operation.

GET-ATTRIBUTES-MATCH is necessary because there will be situations where the contents of the elements of an array may differ, even though the array elements themselves are of the same type. The most obvious example of this is the situation where several network interfaces exist and are of different types, with different data collected for each type.

NOTE: The GET-ATTRIBUTES-MATCH operator will disappear if a generalized filtering mechanism is devised.

ADDITIONAL NOTE: 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 an obvious notation to describe this type of organization. So, we're stuck with the GET-ATTRIBUTES operator. However, if this cleaner organization becomes possible, this decision may be re-thought.

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 array, with an ASN.1 representation of OctetString. Because of the variety of addressing architectures in existence, the conversion between the OctetString and "memory" 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 dictionary. In a one-address-space situation, this dictionary will be at the top level. If each process has its own address space, then one "memory" dictionary may exist for each process.

The GET-RANGE operator is provided for the primary purpose of retrieving the contents of memory, but can be used on any array. It is only useful in these other contexts if the array index is meaningful.

```
GET-RANGE   array start length    GET-RANGE    dict
            Get <length> elements of <array> starting at <start>.
            <start> and <length> are both ASN.1 INTEGER type.
```

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

Memory is special in that it will not automatically be returned as part of a request for an entire dictionary (e.g., If memory is part of the "system" dictionary, then requesting:

```
system{}
```

will emit the entire contents of the system dictionary, but not the memory item).

NOTE: The GET-RANGE operator may disappear if a generalized filtering mechanism is devised.

Controlling Things

All of the operators defined so far only allow data in an entity to be retrieved. By replacing the "template" arguments used in the GET operators with values, data in the entity can be changed.

There are many control operations that do not correspond to simply changing the value of a piece of data, such as bringing an interface "down" or "up". In these cases, a special data item associated with the component being controlled (e.g., each interface), would be defined. Control operations then consist of "setting" this item to an appropriate command code.

SET `dict value SET dict`
Set the value(s) of data in the entity to the value(s) given in <value>.

SET-MATCH `array mvalue svalue SET-MATCH dict`
<array> should be a array (dictionary containing only one type of item). The first tag in <mvalue> and <svalue> must match this type. For each entry in <array>, match the <mvalue> against the contents of the entry. If there is a match, set value(s) in the entity to the value(s) in <svalue>, just as in SET.

CREATE `array value SET dict`
Insert a new entry into <array>. Depending upon the context, there may be severe restrictions about what constitutes a valid <value>.

DELETE `array value SET dict`
Delete the entry(s) in <array> that have values that match <value>.

If there are several leaf items in the matched value, as in
`route-entry{ interface(1), cost(3) }`
all of them must match an array entry for any values to be changed.

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

```
interfaces BEGIN                      -- get dictionary
interface{ ip-addr(10.0.0.51) }       -- value to match
interface{ status(down) }             -- value to set
SET-MATCH
END                                   -- finished with dict
```

Delete the routing table entry for 36.0.0.0.

```
route-table BEGIN                       -- get dictionary
route-entry{ ip-addr(36.0.0.0) }        -- value to match
DELETE
END                                     -- finished with dict
```

Note that this BEGIN/END pair ends up sending an empty ASN.1 item. We don't regard this as a problem, as it is likely that there will be some get operations executed in the same context. In addition, the "open" ASN.1 item provides the correct context for reporting errors. (See page 14.)

NOTE: The SET-MATCH operator will disappear and the DELETE operator will change if a generalized filtering mechanism is devised.

Atomic Operations

Atomic operations can be provided if desired by allowing the stack to contain a fragment of a query. A new operation would take a query fragment and verify its executability and execute it, atomically.

This is mentioned as a possibility, but it may be difficult to implement. More study is needed.

ERRORS

If some particular information is requested but is not available for any reason (e.g., it doesn't apply to this implementation, isn't collected, etc.), it can ALWAYS 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 tag doesn't refer to a dictionary, an ERROR object be emitted. The contents of this object identify the exact nature of the error and are discussed in RFC-1024.

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.

If there was an error before any ASN.1 objects were generated, then the result would simply be:

```
error(details)
```

If a couple of ASN.1 objects were unterminated, the result might look like:

```
interfaces{
     interface { name(...) type(...) error(details) }
     error(details)
     }
error{details}
```

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 are two context-dependent tags assigned by the vendor for its non-standard data.

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

IMPLEMENTATION

Although it is not normally in the spirit of RFCs to define an implementation, the authors feel that some suggestions will be useful

to early 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.

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

- An object-oriented approach should make initial 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.

- 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, MATCH, SET, etc.

- The hardest part is actually extracting the data from an existing TCP/IP implementions that weren't designed with detailed monitoring in mind. This should be less of a problem if a system is designed with easy monitoring as a goal.

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