Desktop Management Interface Specification

DSP0005 **STATUS: Preliminary**

Version 2.0.1s
January 10, 2003

Technical inquiries and editorial comments should be directed in writing to:

Distributed Management Task Force
c/o MKI
54 SW Yamhill St.
Portland, OR 97204
(503) 225-0725
(503) 225-0765 (fax)
email: **dmtf-info@dmtf.org**

Additional copies of this specification may be obtained free of charge electronically via internet at:
ftp//ftp.dmtf.org
or
from the World Wide Web at:
http://www.dmtf.org

CONTENTS

1. INTRODUCTION AND OVERVIEW

1.1 MOTIVATION

Within a computer system, there is a gap between management software and the system's components that require management. Managers must understand how to manipulate information on a constantly growing number of products. In order for products to be manageable, they must know the intricacies of complex encoding mechanisms and foreign registration schemes. This arrangement is not desirable from either side.

This document describes the Desktop Management Interface, or DMI, that acts as a layer of abstraction between these two worlds.

The DMI has been designed to be:

- independent of a specific computer or operating system
- independent of a specific management protocol
- easy for vendors to adopt
- usable locally — no network required
- usable remotely using DCE/RPC, ONC/RPC, or TI/RPC
- mappable to existing management protocols (e.g., CMIP, SNMP)

The DMI procedural interfaces are specifically designed to be remotely accessible through the use of Remote Procedure Calls. The RPCs supported by the DMI include:

- DCE/RPC
- ONC/RPC
- TI/RPC

1.2 BASIC TERMINOLOGY

Throughout this document, ***system*** means a computer system. ***Components*** are physical or logical entities on a system, such as hardware, software or firmware. Components may come with the system or may be added to it. The code that carries out management actions for a particular component is known as the ***component instrumentation***.

A ***management application*** is a program that initiates management requests. A management application uses the DMI to perform management operations. The management application may be a program such as an application with a graphical user interface. It may be a network management protocol agent that translates requests from a standard network management protocol (such as SNMP or CMIP) to the DMI and back again.

DMI Service Provider, which is analogous to the DMI Service Layer of previous DMI specifications, may be shortened to just ***DMI SP*** throughout this document. The abbreviations ***DMIv1.x*** and ***DMIv2*** are used respectively to refer to the DMI 1.x and DMI 2.0 specifications.

Other terms are highlighted in italic bold when first introduced. A full glossary is provided in Appendix E.

1.3 ELEMENTS OF THE DMI

The DMI has four elements:

1. a format for describing management information
2. a service provider entity
3. two sets of APIs, one set for service providers and management applications to interact, and the other for service providers and components to interact.
4. a set of services for facilitating remote communication

Component descriptions are defined in a language called the ***Management Information Format***, or ***MIF***. Each component has a ***MIF file*** to describe its manageable characteristics. When a component is initially installed into the system, the MIF is added to the (implementation-dependent) ***MIF database***.

DMI Service Providers expose a set of entry points that are callable by Component instrumentation. These are collectively termed the ***Service Provider API for Components***. Likewise, Component instrumentation code exposes a set of entry points that are callable by the DMI Service Provider. These are collectively termed the ***Component Provider API***. In the DMI Version 1.x specification, these two APIs were together embodied in the Component Interface.

The ***Component Interface***, or ***CI***, is used by component providers to describe access to management information and to enable a component to be managed. The CI and the MIF shield vendors from the complexity of encoding styles and management registration information. They do not need to learn the details of the popular and emerging management protocols.

Previous versions of this specification defined the CI to be a block oriented data interface as opposed to a procedural interface. This specification introduces a new procedural CI interface. All new functions introduced by this specification are available only as part of the new procedural CI. [1]

> **NOTE** that the functions in the Component Interface are OS-specific. Some OSes may not implement the CI but provide equivalent functionality using other, native mechanisms. In the rest of this document, the use of the term CI should be taken to stand equally for other OS-specific implementations of this functionality.

The DMI Service Provider also exposes a set of entry points callable by Management Applications. These are collectively termed the ***Service Provider API for Management Applications***. Likewise, Management Applications expose a set of entry points callable by the DMI Service Providers. These are collectively termed the ***Management Provider API***. In the DMI Version 1.x specification these were together embodied in the Management Interface.

The ***Management Interface***, or ***MI***, is used by applications that wish to manage components. The MI shields management application vendors from the different mechanisms used to obtain management information from elements within a computer system.

Previous versions of this specification defined the MI to be a block oriented data interface as opposed to a procedural interface. This specification introduces a new procedural MI interface. All new functions introduced by this specification are available only as part of the new procedural MI. [1]

The new procedural MI introduced with this specification is a remotable interface designed to be used with one of the supported RPCs.

The ***DMI Service Provider***, previously called the ***Service Layer*** (SL), is an active, resident piece of code running on a computer system that mediates between the MI and CI and performs services on behalf of each.

A functional block diagram is shown in Figure 1-1.

The DMI Version 1.1 block oriented MI and CI interfaces are local interfaces, to be used within a single system. The new procedural MI introduced with this specification is a remotable interface designed to be used with Remote Procedure Call. The new procedural CI is a local interface, to be used within a single system.

In Figure 1-1 all hardware and software components, the MIF Database, and the DMI Service Provider exist within a single system, or are directly attached, such as printers or modems. The management applications may be command-

[1] The DMTF Compliance Guidelines Document contains the information regarding backwards compatibility of previous DMI specifications (the DMIv1.x block interface in particular).

line or graphical user interface programs, located on the local system or located on remote management work-stations. Network protocol agents may be used to translate between a particular management protocol and the DMI.

Note: It is valid for component instrumentation to register permanently or temporarily as an MI application in addition to a CI registration.. This is usually used by components as a means of dynamically obtaining their current component ID at runtime from the DMI Service Provider.

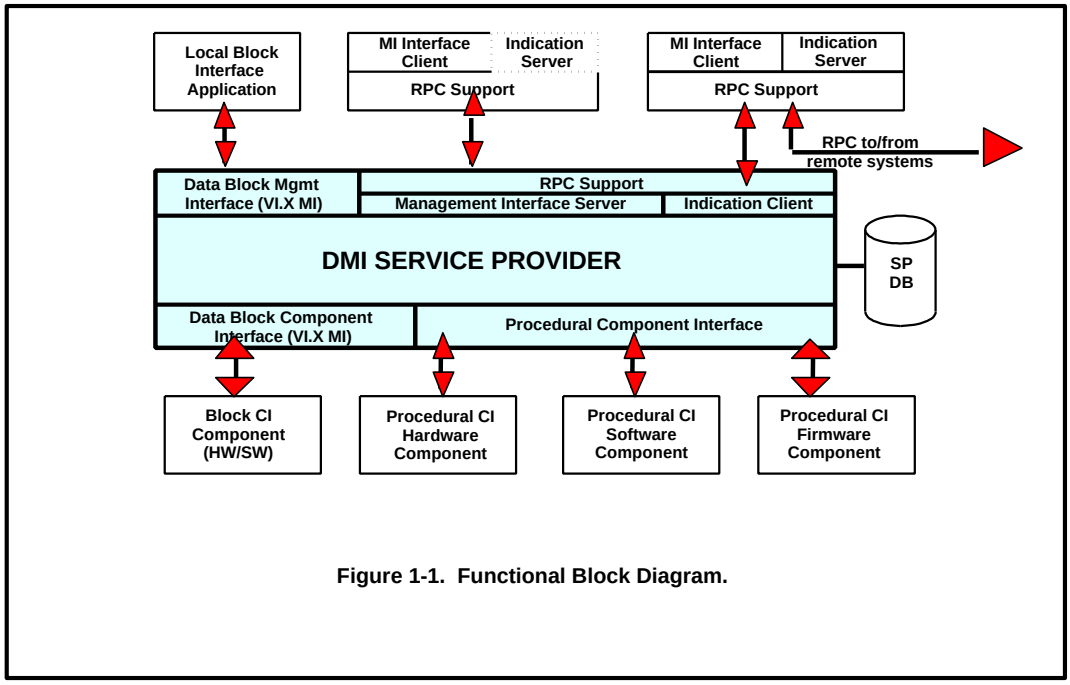


Figure 1-1. Functional Block Diagram.

1.4 DATA MODEL

Components have one or more named ***attributes*** that collectively define the information available to a management application. Attributes are collected into named ***groups*** for ease of reference. Groups may be scalar or may be multiple instantiations, such as the set of attributes for each instance of a network interface table. Multiply instantiated groups are called ***tables***, and a ***row*** (instance) of a table is referred to by a set of attributes that form a ***key***.

So, within a system, there are many components, each with one or more groups. Each group has one or more attributes; and each group may be multiply instantiated as a table. The component instrumentation presents this component/group/key/attribute representation to the management application. A diagram is shown in Figure 1-2.

Component instrumentation may respond to requests by management applications, and may offer unsolicited information (***indications*** or ***events***).

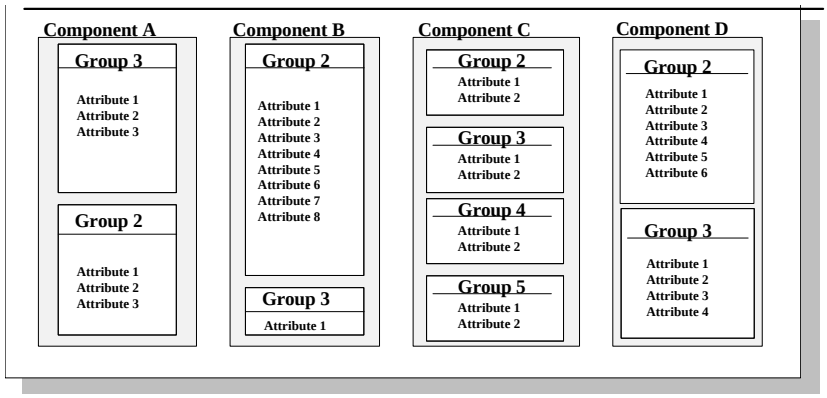


Figure 1-2. Diagram of Attribute Representation In Data Model.

1.5 THE DMI SERVICE PROVIDER

The DMI Service Provider coordinates and arbitrates requests from management applications to the specified component instrumentation's. The DMI Service Provider handles the run-time management of the MI and CI, which includes component installation, registration at both levels, request serialization and synchronization, and general flow control and housekeeping.

The interfaces have been designed so that commands at the MI level are either satisfied at the DMI Service Provider or passed directly to the CI.

Figure 1-3 depicts a possible DMI Service Provider block diagram. This is an example only and is not part of the DMI specification.

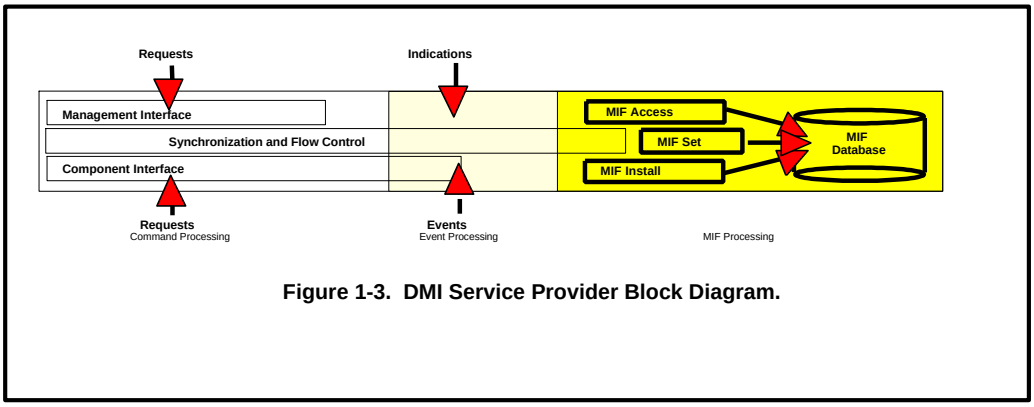


Figure 1-3. DMI Service Provider Block Diagram.

1.5.1 Service Provider Responsibilities

The DMI Service Provider (SP) must coordinate the dynamic installation and removal of component instrumentation's and management applications. It must enforce that at least group 1 (the component ID group) is in each installed .

The DMI SP must coordinate the registration of entities wishing to initiate management activities.

The DMI SP is responsible for all run time accesses to the MIF data. Implementations of the DMI Service Provider may choose to store MIF files in an internal format (a MIF database) for performance and ease of access.

The DMI SP is responsible for launching the component instrumentation, if necessary.

The DMI SP must enforce command serialization to a component instrumentation and ensure that commands are allowed to run to completion. Multiple requests for a particular component instrumentation must be queued.

The DMI SP must support event/indication subscription and filtering.

The DMI SP must forward indications based on subscription and filters to each registered management application, and must time-stamp incoming indications before forwarding them.

The DMI SP must send indications to all registered management applications which have subscribed for indications when components are installed or removed from the MIF database.

The DMI SP must appear to management applications as a component with ID 1 (one). As a component, it must support the standard ComponentID group, defined in Section 3.1.1. Additionally, the DMI SP must support the Subscription Indication and Filter standard groups. Also like any component, it may define additional groups beyond the ComponentID group.

The DMI SP must support all of the NLS mechanisms contained in this specification, including Unicode and multiple NLS installations of schema for each component.

1.6 OPERATIONAL CHARACTERISTICS

The relationship among management applications, the DMI Service Provider and component instrumentation can exist as a many-to-one-to-many relationship. There may be many management applications issuing commands through a single DMI SP to manage many components. If multiple management applications are active, each by have a different language specified, requiring component instrumentation to support multiple languages simultaneously.

For purposes of identification, management applications must register with the DMI SP before they can participate in management functions. Component instrumentation's must install into the DMI SP once when first introduced to the system. Components implemented using the Direct Interface MUST register with the DMI SP when they wish to notify it of their immediate availability. The mechanics of "connecting" to the DMI SP to register or issue commands may differ among operating systems and DMI SP implementations.

Control flow is usually initiated from the management application to the DMI Service Provider and on to the component instrumentation. There may also be *indications*, which are unsolicited reports that flow in the opposite direction.

There are three general categories of access commands: Get, Set and List. The Get and Set commands let management applications read and write manageable entities within a system.

The List commands return "meta" information; information about the component MIF itself. The List commands do not get the actual attribute values within the component. List commands allow a management application to get the semantic information in a MIF. Since the DMI Service Provider gets MIF information from its MIF database, the List commands do not cause any component instrumentation code to be invoked.

Along with these standard access commands are commands to register/unregister management entities, and allow component instrumentation's to generate indications.

Within DMI data structures, all strings are stored in the form <*length*> <*data*>, where <*length*> is an unsigned 32-bit value giving the number of *octets* in the <*data*> part of the string. Note that the number of *characters* in the string depend on whether it is in ISO 8859-1 format (1 octet/character) or Unicode format (2 octets/character. In DMIv1.x, String <data> values were not required to be zero-terminated as in the C programming language. For DMIv2.0, they must be NULL terminated in addition to the <length> specifier.

Component instrumentation's are serially re-usable, but they are not expected to be re-entrant.

The DMI does not provide primitives to own or lock resources over a sequence of commands. Multiple management applications may make simultaneous accesses to the interfaces described in this document. Grouping and scheduling of operations, other than the synchronization provided by the DMI Service Provider, are the responsibility of the management application. Likewise, any desire for mutual exclusion, to lockout certain accesses, or to provide DMI database security in any form, is the responsibility of the management application.

1.7 REMOTEABLE INTERFACE

The Data Block interface introduced in April of 1994 with DMI version 1 (DMIv1.x) uses a single entry point ('DmiInvoke') and is passed a set of concatenated data structures. At the time DMIv1.x was created, it was felt that this type of interface was needed for low level access such as when crossing protection rings in a protected processor, interfacing to device drivers, and for easy packaging when remoting. The remoteable interface presents a procedural interface as opposed to DMIv1.x's block oriented interface. The procedural interface, in addition to being suitable to remoting via one of the supported RPC mechanisms defined previously, is much friendlier to programmers and much less error-prone.

RPC issues are limited to the opening and closing of remote sessions. Network-centric issues like transports, name resolution, etc. are provided by the RPC services used and are outside of the scope of this specification.

The remotable interface (DMIv2.0) is designed to provide remote access to DMI functionality and data while hiding the intricacies of manipulating the DMIv1.x data blocks. DMIv1.x often 'batches' together somewhat related functions into single commands. This results in commands which return lots of related information and requires the caller to pull out what they want. In DMIv2.0, calls are broken out functionally to provide specific information. Therefore a given DMIv1.x command may equate to multiple DMIv2.0 commands, each one performing a specific function.

RPC is based on a client / server architecture. The client side includes a set of ***Stubs*** which have interfaces with the same signatures as the function calls they represent on the server. The stubs interact with the local RPC support to exchange the input parameters, the output parameters, and return codes with the remote procedure located at the server. A Remote node acts as a client for procedural MI function calls, and a server when receiving indications. The node under management acts as a server for procedural MI function calls, and as a client when delivering indications to a remote node.

Figure 1-4 shows the overall architecture for the remoteable interface. Note that the CI is a local interface and is not remoted. Specific implementations of this specification may vary somewhat in the actual structure of the software elements as shown.

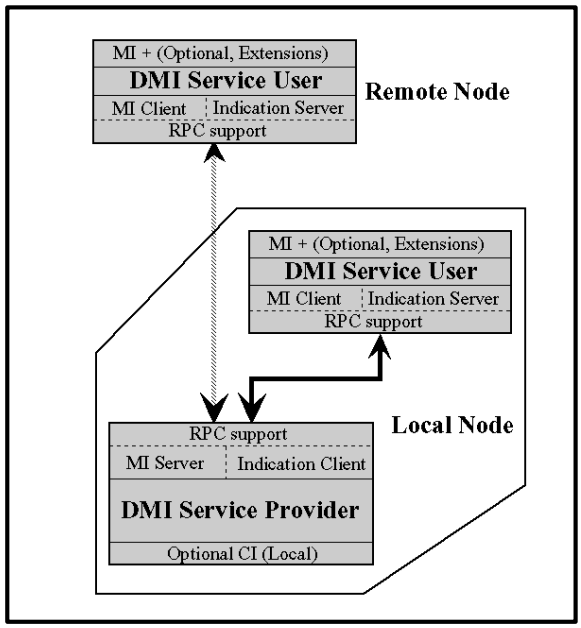


Figure 1-4. Remotable Interface Architecture

Certain elements of DMIv1.x are not present in DMIv2.0. The concept of concatenated command blocks has been removed in DMIv2.0. DMIv2.0 is a totally synchronous call interface whereas DMIv1.x is asynchronous. Link level security, new to DMIv2.0, is provided using the underlying RPC security mechanism.

1.8 SECURITY

DMIv2.0s defines a mechanism to control remote access to the ***DMI*** Management Interface and local access to ***DMI*** interfaces. The remote access control mechanism is defined on top of standard RPC mechanisms, whereas the local access control mechanism is defined on top of operating system mechanisms. ***DMIv2.0s*** does *not* specify a standard format for identities nor a cryptosystem to verify those identities, but relies on those provided through the RPC and by the operating system. The main features introduced by ***DMIv2.0s*** are authentication, role-based authorization, flexible policy, security indications and logging. ***DMIv2.0s*** is an extended version of ***DMIv2.0*** specification. The bulk of the DMI Security Extension appears in Sections 10 through 18.

The DMI Security Extension is conditionally required. That is, if a DMI Service Provider implementation provides an access control mechanism, it has to implement the DMI Security Extension as defined in this specification.

Note that DMI2.0s security is based on the security infrastructure provided by the RPC and the Operating System. Therefore, if the security of the RPC or the Operating System is compromised, DMI2.0s security will be compromised as well. For example, if a malicious user can circumvent the file system security and modify the MIF database on a system, she could modify the DMI2.0s policy in the database to her advantage.

2. INFORMATION SYNTAX

2.1 MANAGEMENT INFORMATION FORMAT

Managed information is described in a simple format called the Management Information Format, or MIF. The MIF defines components and their associated attributes. Files that contain information structured to MIF guidelines are known as MIF files. Each instance of a managed component must provide a separate MIF file that describes the manageable aspects of that component.

The MIF file is a text file that is "installed" -- presented to the DMI Service Provider for inclusion in the MIF database. Modifications to the MIF file can be made with a text editor, although component providers are encouraged to automate this process.

This section describes the MIF. The complete BNF syntax is specified in Section 2.2. A sample MIF file is given in Section 2.3

2.1.1 Lexical conventions

The MIF uses either the International Standards Organization document ISO 8859-1 (Latin Alphabet no. 1) or Unicode 1.1 specification for its character sets. If a Unicode MIF is provided, the first octet of the MIF file must be 0xFE (hexadecimal), and the second must be 0xFF. Otherwise the DMI Service Provider will treat the file as an ISO8859-1 MIF.

There are four classes of tokens: keywords, integer constants, strings (literals), and separators. Two keywords, **start** and **end**, are scope keywords that are only useful when followed by another keyword. Blanks, tabs, new lines, carriage returns and comments (collectively, "white space") as described below are ignored except as they serve to separate tokens. White space is required to separate otherwise adjacent keywords and constants.

The MIF is case insensitive in all cases except for literal strings (characters surrounded by double quote characters), where case is retained.

Literal strings separated by white space are concatenated and stored as one literal string.

2.1.2 Comments

Comments may be placed throughout the file, and are ignored. The start of a comment is denoted by two consecutive forward slashes ("//"). The comment continues through the end of the line.

2.1.3 Keywords

The MIF uses the following keywords:

component	group	attribute
table	path	enum
name	description	id
type	class	key
value	access	storage
language	start	end
unsupported	counter	counter64
gauge	octetstring	displaystring
string	integer	int
date	integer64	int64
win16	win32	dos
macos	os2	unix
read-only	read-write	write-only
direct-interface	common	specific
pragma	win9x	winnt
unknown		

2.1.4 Data types

The MIF supports data types that describe the storage requirements as well as some semantics. The type can be:

DAT \ TYPE	DESCRIPTION
integer (or int)	A 32-bit signed integer; no semantics known
integer64 (or int64)	A 64-bit signed integer; no semantics known
gauge	A 32-bit unsigned integer that may decrease or increase
counter	A 32-bit unsigned integer that never decreases
counter64	A 64-bit unsigned integer that never decreases
string (n) or displaystring(n)	A displayable string of n octets Note: For 8859-1, 1 octet/character; For Unicode, 2 octets/character
octetstring(n)	A string of n octets, not necessarily displayable
date	A 28-octet displayable string, described below

A counter increases to its maximum value (2^{32}-1 or 2^{64}-1) and rolls over to zero at its maximum value. An automobile's odometer is an example of a counter.

A gauge may increase or decrease, but when it reaches its maximum value (2^{32}-1), it continues to report the maximum value until the value decreases below the maximum. An automobile's speedometer is an example of a gauge.

For the string types, the declared length *n* represents the maximum number of octets in the string. The actual number of octets in use may be shorter than this maximum value. **displaystring**s are required to be zero-terminated as in the C/C++ programming languages. String lengths represent the number of *octets* in the string for **displaystring**s and include the terminating null character (Note, that in the case of Unicode a null character is 2 octets). In the case of **octetstring**s the length *n* is the number of octets in the string.

Implementation notes:

1) In the implementation of the string types the actual length of the string is computed and stored as part of the string datastructure. See Section 5.3 for details.

2) Attributes whose values are Strings, OctetStrings, or DisplayStrings are required by the MIF syntax to specify a maximum string length as part of their definition. However, in certain resource constrained environments, it is possible that component instrumentation for such an attribute may implement a smaller maximum length for the attribute. Therefore, consumers of MIF information must first ascertain the implemented maximum length of a string attribute before operating on it, regardless of what the published MIF definition of the attribute might state. This may be done through the use of the DmiListAttributes entry point that is defined in Section 6.2.6.

Dates are defined in the displayable format

```
yyyymmddHHMMSS.uuuuuu+ooo
```

where `yyyy` is the year, `mm` is the month number, `dd` is the day of the month, `HHMMSS` are the hours, minutes and seconds, respectively, `uuuuuu` is the number of microseconds, and `+ooo` is the offset from UTC in minutes. If east of UTC, the number is preceded by a plus (+) sign, and if west of UTC, the number is preceded by a minus (-) sign. While this is only 25 octets, the date is stored as a 28-octet field for memory alignment reasons, and the last three octets are zero (‘\0’).

For example, Wednesday, May 25, 1994, at 1:30:15 PM EDT

would be represented as: `19940525133015.000000-300`

Values must be zero-padded if necessary, like "05" in the example above. If a value is not supplied for a field, each character in the field must be replaced with asterisk ('*') characters.

2.1.5 Constants

Integer values may be specified as in the C/C++ programming languages:

SYNTAX	*BASE*
nnn	*decimal*
0nnn	*octal*
0xnnn or 0Xnnn	*hexadecimal*

where *n* is a digit in the proper base.

The MIF does not support floating point values.

Literals (strings) are character sequences surrounded by double quotes. Adjacent double quote characters (besides white space) indicate multi-part literals that are treated as one string. For example:

```
"This is an example"
" of a multi-part"
" literal string."
```

The literal escape character is the backslash. It is used as in C/C++, to enter the following characters:

SEQUENCE	*CHARACTER*
\a	*alert (ring terminal bell)*
\b	*backspace*
\f	*form feed*
\n	*new line*
\r	*carriage return*
\t	*horizontal tab*
\v	*vertical tab*
**	*backslash*
\"	*double quote*
\xhh	*bit pattern, hexadecimal*
\ooo	*bit pattern, octal*

For the octal bit pattern, *ooo* can be one, two or three octal digits (from \0 to \377) when the MIF is specified in ISO8859-1 format, and from one to six octal digits (from \0 to \177777) when the MIF is in Unicode format.

For the hexadecimal bit pattern, *hh* can be one or two hex digits (from \x0 to \xff) when the MIF is specified in ISO8859-1 format, and from one to four hex digits (from \x0 to \xffff) when the MIF is in Unicode format.

If the character following a backslash is not one the letters specified in the above table, the backslash is being used as a quoting character. This use of the backslash is necessary to quote characters in those situations where those characters might otherwise trigger inappropriate syntax processing to occur e.g. the inclusion of a "" (double-quote) character in a string is not possible without quoting, since "" characters are used to delimit strings.

The rules for using the '\' (backslash) character as a quoting character are as follows :

- Any printing character other than a,b,f,n,r,t, and v, may be quoted by prefacing it with the '\' character. In particular '\' may be used to quote itself by using '\\'.
- In nested strings, the characters in the inner strings that might interfere with the parsing of the outer string must be quoted
- If strings are nested more than two deep, then the quoting character must itself be quoted a number of times that is equal to the nesting depth minus one. e.g.

```
"This is a first level string containing \"A second level string"
" and \\\"a third level string\\\"\""
```

In this example the "" characters quoting the second level string are quoted. In the third level string the '\' character that quotes the "" characters must itself be quoted as '\\'.

- Non printing characters must be provided by their escaped octal or hexadecimal forms as described above.

2.1.6 Block scope

The keywords **start** and **end** delimit the scope of a definition block. An associated keyword must follow both **start** and **end**. The keywords and their scope are:

BLOCK	WITHIN	DESCRIPTION
component	MIF file	defines a component. All other blocks exist within this scope. There can be only one component definition per MIF file.
path	component	associates a symbolic string with operating system-specific path names. Zero or more path definitions may exist in the MIF, usually at the top of the file before any groups.
group	component	defines a collection of attributes, sometimes used as a template row for a table. At least one group is required per MIF file (the ComponentID group, defined below).
attribute	group	defines a unit of managed data. All attributes "exist" within the scope of a group definition. A group must have at least one attribute in it.
table	component	defines one or more instances of a group using a previously defined group. Optional.
enum	component or attribute	defines a list of integer-to-string mappings. Named enumerations can be defined at the component level, while unnamed enumerations can be defined within the scope of an attribute definition. Optional (but while many enum definitions can exist at the component level, only one can be defined per attribute)

Here's an example of the structure of a MIF file. For readability, only one of each block is given. Each level is indented for readability:

```
start component
        start path
        end path
        start enum
        end enum
        start group
                    start attribute
                        start enum
                        end enum
                    end attribute
        end group
        start table
        end table
end component
```

2.1.7 Language statement

The **language** statement is used to describe the native (human) language of the MIF file. This statement appears before the **start component** statement. The syntax is

```
language = "language string"
```

where *language string* is a text string that identifies the language, dialect (as territory) and character encoding. The format of language string is:

```
language-code|territory-code|encoding
```

where *language-code* is one of the two-letter codes defined in ISO 639, *territory-code* is one of the two letter codes defined in ISO 3166, and *encoding* is either **iso8859-1** or **unicode**. For example, the language string:

```
"fr|CA|iso8859-1"
```

indicates French Canadian, with ISO 8859-1 (8-bit) encoding.

If any fields are not supplied, they are simply omitted, but the two vertical bars must appear in the string. The default language string is "`en|US|iso8859-1`".

The *encoding* field is ignored in the MIF file because the first two bytes of the file determine the encoding. However the field is used when communicating through the MI.

The language statement may appear only once per MIF file.

Samples of the codes defined in the two ISO standards are in Sections 2.4 and 2.5.

A note on localization: MIF files that have been translated (localized) should translate only literal strings such as names, descriptions and enumeration literals, and any comments within the MIF. Neither class strings nor language names may be localized. Keywords must not be localized.

2.1.8 Common statements

The following three statements can be used within the scope of most definitions, as noted. Definition-specific statements are described when the definition is described.

2.1.8.1 NAME STATEMENT

The required **name** statement is used inside the scope of a definition to assign a relatively short string to the definition. The name is normally used for display to users, and must be less than 256 characters. The syntax is:

```
name = "name string"
```

where *name string* is defined by the MIF file provider. However, users may edit the MIF file and change the name. The name statement may appear only once per definition. Names are not required to be unique except for enumeration and path names, which must be unique among other enum (and path) names within a component.

2.1.8.2 DESCRIPTION STATEMENT

The optional **description** statement is used inside the scope of a definition to give more information about the element being defined. The description is used for display to users. The syntax is:

```
description = "description string"
```

where *description string* is defined by the MIF file provider. However, users may edit the MIF file and change the description.

The description statement is used in the component, group and attribute definitions. The description statement may appear only once per definition.

2.1.8.3 ID STATEMENT

The **id** statement is used inside the scope of a definition to assign a unique numeric identifier for the definition. Each type of definition that is required to have an id must have a unique id within its scope. The id is used for naming items at the API level, and for mapping to network management protocols. The syntax is:

```
id = n
```

where *n* is defined by the MIF file provider. The value of *n* must be a non-zero 32-bit unsigned integer, and must be unique within the scope of the containing definition. For example, all attributes within a group must have different IDs, but attribute IDs do not need to be unique across groups. Since components and management applications use these IDs for communication, users may not change them.

The id statement is required in the attribute and table definitions. It is optional in the group definition. It is not used in the component, path and enum definitions. While components have IDs, they are assigned by the DMI Service Provider at installation time. The id statement may appear only once per definition.

2.1.9 Component definition

The component definition has the following syntax:

```
start component
         name = "component name"
         [description = "description string" ]
         [pragma = "pragma string" ]
         (component definition goes here)
end component
```

Only one component definition may appear in a MIF file.

2.1.10 Path definition

Path definitions are used to locate the files used for active management of the component. The definition begins with the statement **start path**, followed by a **name** statement that defines a symbolic name, and a number of lines equating

operating system identifiers to the path of the callable program. The symbolic name may be used later in attribute definitions, indicating that the value for the specified attribute should be retrieved or set by invoking the associated callable function. The path definition ends with the keyword **end path**.

The operating system identifiers are **dos**, **macos**, **os2**, **unix**, **win16, win32, win9x, and winnt**. Case is not significant.

NOTE: Use of the Win32 keyword implies that the instrumentation in question will function on either Windows 9x or Windows NT. Using the specific keywords: win9x or winnt implies that the component will ONLY run on that environment.

If the component instrumentation is provided by code that will connect to the DMI Service Provider (as opposed to having the SL start the code at request time), the keyword **direct-interface** may be supplied instead of a path name.

Here's an example:

```
start path
    name  = "Performance Info Instrumentation Code"
    win16 = "C:\\someplace\\wincode.exe"
    os2   = "C:\\someplace\\os2code.dll"
    dos   = "C:\\someplace\\doscode.ovl"
    unix  = direct-interface
end path
```

Many path definitions may appear within the component definition; potentially one for each callable function. The path name must be unique among all other path names in this component definition.

See the sample MIF (Section 2.3) for usage of the symbols defined in the path definition.

2.1.11 Enum definition

Enumerated lists allow strings to be associated with signed 32-bit integers. They are defined within the component scope or within the scope of individual attributes. These enumerations are primarily used by component instrumentation to pass integers through the DMI, so management applications can display the corresponding text string in the user's native language.

The syntax of enumerated lists is:

```
start enum
         name = "enum name"
         vvv = "string literal for vvv"
         [xxx = "string literal for xxx"]
end enum
```

"*enum name*" is a unique enumeration list name within this component.

Integer values *vvv* and *xxx* above can be listed in any order and do not have to have every number represented between the lowest and highest listed. However each value must be unique within this enumeration definition.

Many enum definitions may appear within the component definition; one for each enumeration list. Enumerations do not have **id** or **description** statements.

2.1.12 Group definition

A group is a collection of one or more attributes. Groups let component providers arrange attributes into logical sets. Groups can also be used to represent arrays (tables) of attributes. The use of groups allows logical subsets within a component to be standardized across vendors.

The syntax of a group definition is:

```
start group
        name = "group name"
        class = "class string"
        [id = nnn]
        [description = "description string"]
        [key = nnn[,mm]...]
        [pragma = "pragma string"]
        ( attribute definitions go here)

end group
```

The **id** statement, if provided, must have a value unique among other groups within the component. Specifying a group id without a key means that this group definition defines a group. If both **id** and **key** are provided, the group definition represents a table but that group is not necessarily supported by component instrumentation code. Groups that provide both an **id** and **key** can be used again later as a template in the creation of a table.

If the **key** statement is provided and the **id** statement is not provided, the group definition represents a template row in a to-be-defined table, and the **value** statements (defined below) refer to default values within the row. A **table** definition may follow to populate the table based on the template. See the section 2.1.16 on table definition for more.

The following table describes the possibilities:

KEY?	ID?	RESULT
No	No	error
No	Yes	scalar group (not a table. Id is the group's ID)
Yes	No	template (table definitions may follow)
Yes	Yes	table (Id is the table's ID. Can be used as a template later)

Many groups may be defined within the component.

2.1.12.1 CLASS STATEMENT

The required **class** statement is used inside a group definition to identify the source of the group and the group version. All groups using the same class string ***must*** share the same attribute definitions within the group, including attribute type, access, storage (defined below) and IDs. The attribute name, description and value may be different, however. This assists management applications in determining the semantics of the group's attributes. Groups are identified as unique only by their class string, not their Group ID. So management applications must retrieve the allocated ID of a group by using its unique class string in a List command (refer to Section 6).

The class statement syntax is:

```
class = "class string"
```

where, by convention, *class string* is encoded as

```
"defining body|specific name|version"
```

In this convention, *defining body* is the name of the organization (such as "DMTF", "IEEE", "Acme Computer", etc.) defining the group; *specific name* identifies the contents of the group ("Server Stats", "Toaster Controls", etc.) and *version* identifies the version of the group definition (001, 002, 003 etc.).

Essentially the class string is an opaque string, and any convention may be used. However, since applications and DMI Service Providers might rely on this convention for obtaining information via the List Component command, component providers are encouraged to use this convention.

It is an error to specify the same class string for two groups if the group definitions are different. Management applications can count on identical group definitions for identical class strings.

Note that `"DMTF|Sample|001"` is not the same as `"DMTF | Sample | 001"` as one has spaces around the vertical bars and the other does not.

Implementations that provide a subset of the attributes defined by a class must use the **unsupported** keyword within the attribute definition (defined below).

Only one class statement is allowed per group.

2.1.12.2 KEY STATEMENT

When the attributes in a group define a row in a table, the group definition must contain a **key** statement to define the attribute ID(s) that is (are) used as the index into the table. Attributes that act as keys may be of any data type. Keys always identify no more than one instance of a group (row of a table).

The key statement syntax is:

```
key = n[,m]
```

where n is the attribute ID that acts as the key for this table. If multiple attributes are used to index a table, they should be specified as comma-separated integers. When management applications send requests or component instrumentation's send results, key values must be sent in the order that they are listed in the key statement.

Only one key statement is allowed per group.

2.1.13 Pragma statement

Pragma definitions are used to provide additional information about the Component, Group or Attribute. As far as the DMI Service Provider is concerned the <MIF Literal> which is the value of the pragma is simply an opaque octet string. However, by DMTF convention the content of the octet string is structured in the following way:

```
<Pragma String>     ::=
  '"'  { <Pragma Keyword> ':' <Parm> { ',' <Parm> }* ';' }*  '"'
```

where <Pragma Keyword>, and <Parm> contain any literal character allowed by Unicode or ISO 8859-1, EXCEPT the characters ':', ',', ';', '|' and '"' in any encoding unless inserted in the string as

- their quoted forms i.e. '\:', '\,', '\;', '\|' and '\"' respectively, OR
- their escaped hex or octal bit pattern equivalents i.e. in the form *nnn* where the *n*'s are octal digits, or **x**mm* where the *m*'s are hexadecimal digits.

At this time four <Pragma Keyword>s are defined, namely:

SNMP: This keyword takes a value that is an SNMP OID of the form n.n.n.....n.n, where the n's are positive integers. It is intended to help in the DMI-SNMP translation process. This Pragma keyword has meaning only in the context of a Group definition.

Dependent_Groups: This keyword takes a comma-delimited list of one or more class strings as its value. It has meaning only in the context of a Group definition. The class strings in the value of this keyword identify the other Groups that must be implemented for this Group to be functional or meaningful. The class strings that are provided as values for this keyword may have null (wild-carded) portions. For example, in a typical case, a null *version* field implies that the dependency exists on any groups with the same *defining body* or *specific name* portions of the class string.

Implementation_Guideline: This Pragma keyword may take one of the three following values: REQUIRED, OPTIONAL, or OBSOLETE. It has meaning only in the context of a DMTF Standard Group definition.

- The value REQUIRED indicates that the working committee that defined this standard group thought it important that it be implemented.
- The value OPTIONAL indicates that the working committee that defined this standard group wished to allow implementors the option of not implementing it.
- The value OBSOLETE indicates that the working committee that defined this standard group recommends that new products should implement the new group that replaces this group, other than this group which has been superseded.

NOTE: This does not invalidate implementations of this group that are already in the field. Management Apps will have to continue to recognize and utilize this obsolete group as well as its successor.

Here is an example of a Pragma statement in a Group definition:

```
start group
name = "ABCD"
class = "DMTF|ABCD|001"
...
```

```
...
pragma = "Dependent_Groups:\"DMTF|FRU|\"; "
         "Implementation_Guideline:REQUIRED;"
...
end group
```

This example pragma definition states that the dependent group for DMTF Standard Group "ABCD" has the class string "DMTF|FRU|". This means that implementing the group "ABCD" is not meaningful unless the group represented by "DMTF|FRU|" has also been implemented. Note that the version number of the dependent group has been wild-carded and that the "" and the '|' characters were quoted using '\'. Furthermore, the Implementation_Guideline states that the DMTF working committee, which defined group "ABCD", felt that it was required for implementation

Reg_Key: The syntax for this keyword is as follows:

```
Reg_Key : <Reg_Key_Value> ;
where
<Reg_Key_Value> ::= <Reg_Key_Parm> <MIF Literal>
<Reg_Key_Parm> ::=
     REG_VALUE  |  REG_DLL  |  REG_VXD  |  REG_NONE
<MIF Literal> ::= <as defined in the MIF grammar>
```

The `<MIF Literal>` field may be any legal, properly constructed, embedded string in the form prescribed by Section 2.1.5 (Constants). In other words, the characters ':' (colon), ',' (comma), and ';' (semi-colon) must be properly quoted, if they occur, by using the '\' (backward slash) character.

The `<Reg_Key_Parm>` field may take one of the four following values: REG_VALUE, REG_DLL, REG_VXD, or REG_NONE.

- The value REG_VALUE indicates a value link to an existing data provider.
- The value REG_DLL indicates a value link to a dynamic link library data provider.
- The value REG_VXD indicates a value link to a dynamic device data provider.
- The value REG_NONE indicates that a value link should not be generated for this attribute.

The value of the Reg_Key pragma is intended to help in the MIF-to-Registry translation process in the Microsoft Windows environment. It is used to provide an indirect value link into the Registry when an attribute value is provided by instrumentation. For further information on this Pragma Keyword, and its usage, please refer to the latest Microsoft documentation. This pragma has meaning only in the context of an Attribute definition.

2.1.14 Attribute definition

An attribute is a piece of data related to a component. Attributes are defined within the scope of a group. The syntax of the attribute definition is:

```
start attribute
name = "attribute name"
id = nnn
[description = "description string"]
type = datatype
[access = method]
[pragma = "pragma string"]
[storage = storagetype]
[value = [v | * "name" | "enum string"
|unsupported | unknown ]]
end attribute
```

The required **id** statement must have a value that is unique among all other attributes within the group.

Groups must have at least one attribute definition. Many attribute definitions may appear within the group definition.

2.1.14.1 TYPE STATEMENT

The required **type** statement in the attribute definition describes the storage and semantic characteristics of the attribute being defined. The syntax is:

```
type = datatype
```

where *datatype* is usually one of the data types previously defined in Section 5.

A data type may be an enumeration; stored and treated as a signed 32-bit integer. Enumerations that have been previously defined (at the component level) can be referenced by name as if they were a type, for example: `type = "Color"`. Enumerations may also be constructed "in line":

```
type = start enum
         (enum definition)
       end enum
```

In this case the enumeration does not need a name since it cannot be referred to outside the scope of this attribute definition. Any name given is ignored.

Only one type statement may appear within the attribute definition.

2.1.14.2 ACCESS STATEMENT

The optional **access** statement determines whether the attribute value can be read or written. The syntax is:

```
access = method
```

where *method* may be **read-only**, **read-write**, or **write-only**. If the access statement is not specified, the default access is **read-only**. Attributes marked as keys may not be write-only. Only one access statement may appear in the attribute definition.

2.1.14.3 STORAGE STATEMENT

The optional **storage** statement provides a hint to management applications to assist in optimizing storage requirements. The syntax is:

```
storage = where
```

where may be **common** or **specific**. **Common** signifies that the value of this attribute is typically limited to a small set of possibilities. An example of **common** may be the clock speed of a CPU. **Specific** signifies that the value of this attribute is probably not a good candidate for optimization because there may be a large number of different values. An example of a **specific** attribute would be a component's serial number.

If the storage statement is not specified, the default storage is **specific**. Only one storage statement may appear in the attribute definition.

2.1.14.4 VALUE STATEMENT

The **value** statement provides a value or value access mechanism. The syntax is:

```
value = v
value = "enumeration value"
value = * "Name"
value = unsupported
value = unknown
```

The value *v* is for read-only attribute values that never change, such as the manufacturer of a component, or for read-write attributes that the DMI Service Provider will handle, as opposed to the component instrumentation. It is illegal to specify *v* for write-only attributes. It must be specified in the correct data type for the attribute; for example dates and literal strings must be specified within double quotes.

The value "*enumeration value*" (a text string enclosed in double quotes) is an enumeration text string that the DMI Service Provider will map to an integer. The mapping must have been previously defined in an **enum** definition within this component or attribute definition, and the attribute's type must be an enumeration. Note that specifying an integer for an enumeration is acceptable.

When reading an enumerated value, there is no guarantee that a mapping exists for that value. Both static and dynamic (instrumented) values may be outside the range of known mappings. This means that Management Applications looking for a mapping must be prepared for the case where the mapping does not exist, and take appropriate action. For example, an application may choose to display the string representation of the enum value. Note: in general it is not considered good practice to return enumerated values that are outside the known range of values, since this reduces the semantic value of the enumerated type.

The value * "*Name*" (a name with "*" before it and surrounded by double quotes) indicates the symbolic name of the component instrumentation code to invoke to read or write the attribute at run time. The symbolic name must have been previously defined in a **path** definition within this component definition.

The value **unsupported** (a reserved keyword) can be given to tell the DMI Service Provider that this attribute is not supported by this component.

The value **unknown** (a reserved keyword) can be given to tell the DMI Service Provider that this attribute is normally supported, but currently unknown.

The value statement is required except when defining table templates, in which case it is optional. If a value is provided within a template, it becomes the default value when populating the table. If it is not provided, there is no default value.

2.1.15 Group example

Here's an example of a group with two attributes:

```
Start Group
     Name  = "Software Template"
     Class = "DMTF|Software Example|001"
     Key   = 1      // key on Product Name
     Pragma      = "SNMP:1.2.3.4.5.6"
     Start Attribute
               ID          = 1
               Name        = "Product Name"
               Description = "The name of the product"
               Storage     = Common
               Type        = String(64)
     End Attribute
     Start Attribute
               ID          = 2
               Name        = "Product Version"
               Description = "The product's version number"
               Type        = String(32)
               Value       = ""
     End Attribute
End Group
```

In this example, the group is acting as a template, because there is no group id and because a key is specified. The default value for the version is an empty string. There is no default for the product name.

2.1.16 Populating tables

An array of group instances is considered a *table*. The instances are *rows* of the table. Often simply defining the group with a key is sufficient for defining the table, since the values of the attributes within each row are provided by the component. However, sometimes it is useful to provide the table's values within the MIF file itself, just as it is sometimes useful to define values within an attribute definition.

The table population mechanism separates the *definition* of the group from the *data* in the group. It uses a previously defined group as a template to store values into the MIF database. The syntax to populate tables is:

```
start table
        name = "table name"
        id = nnn
        class = "class string"

        { v1[,v2 ...] }
        [ { vn[,vm ...] } ]

end table
```

A **name** statement must be supplied that describes this table. The required **id** statement specifies an integer value unique across all other groups and tables within this component. The required **class** statement identifies the previously defined group that is being used as a template.

A group definition specifying both an ID and a Key list defines an empty (zero row) table. The value statements on the attribute definitions do not implicitly define a table row. To initialize a table in the MIF grammar, use the MIF table statement, as described in this section.

Within a table row, the values are provided as in Section 2.2 separated by commas and surrounded by the curly braces "{" and "}". The list of values is provided left-to-right in attribute-ID order; the value of the attribute with the lowest ID appearing first. If a value within the list is omitted, the corresponding attribute value, if defined in the template, is used as the "default" value. It is illegal to omit an attribute's value when no default value was provided in the template. Rows with too few commas are treated as rows with the requisite number of trailing commas, so the values specified in the template are used for the remaining attributes in the row.

Here's an example of populating a table using the group defined in Section 2.1.15.

```
Start Table
        Name    = "Software Table"
        Class   = "DMTF|Software Example|001"
        Id      = 42
        {"Circus", "4.0a"}
        {"Disk Blaster", "2.0c"}
        {"Oleo", "3.0"}
        {"Presenter", "1.2"}
End Table
```

In this example, the resulting table has four rows. The **value** statements in the group definition are used as default values during row population and not as a row themselves.

It is an error to populate rows without providing unique values for the combination of attributes that comprise the key. DMI Service Providers must reject a MIF that does not provide unique keys during row population.

A table definition must come after the group definition to which it refers. The group must have been specified with a **key** statement, and without an **id** statement. More than one table may be created from a single template but each table must have a different id.

2.2 MIF GRAMMAR

The MIF grammar below is expressed in BNF notation, based on the following rules:

1. Items are enclosed in less than and greater than symbols ("<>").
2. An item is defined in terms of other items by identifying the item (<item>), using the symbols "::=" and following with a list of one or more other items (<item1> ::= <item2>).
3. Items not inside of less than and greater than signs ("<>") are considered literals and entered exactly as they are defined. Single character literals are enclosed in apostrophes (').
4. An item enclosed in brackets ("[]") is optional.
5. An item enclosed in braces with an asterisk ("{}*") is present one or more times.

The MIF grammar is as defined as follows:

```
<MIF Source File>  ::=  <Language>  <Component Definition>

<Language>                                ::= Language '=' <Language String>

<Language String>  ::= <MIF Literal>

<Component Definition>                    ::= Start Component
                                              <Component Identification>
                                              <Component Body>
                                              End Component

<Component Identification>                ::= Name '=' <Component Name>

<Component Name>   ::= <MIF Literal>

<Component Body>   ::=[ <Description> ]     |
                                                   [ { <Path Definition > }* ] |
                                                   [ { <Global Enumeration Defn> }* ]  |
                                                   { <Group Definition> }* |
                                                   [ { <Table Definition>  }* ]  |
                                                   [ <Pragma Statement> ]
                                              (Note: These statements may be in any order.)

<Description>                             ::= Description '=' <Description Text>

<Description Text> ::= <MIF Literal>

<Path Definition>  ::= Start Path
                                              <Path Identification>
                                              <Path Body>
                                              End Path

<Path Identification>                     ::= Name '=' <Instrumentation Symbolic Name>

<Instrumentation Symbolic Name>           ::= <MIF Literal>

<Path Body>                               ::= <Path Body> <Path Statement> |
                                              <Path Statement>

<Path Statement>   ::= <OS Name> '=' <Path Value> |
                                              <OS Name> '=' Direct-Interface

<OS Name>                                 ::= DOS | MACOS | OS2 | UNIX | WIN16 | WIN32 |
                                              WIN9x | WINNT

<Path Value>                              ::= <MIF Literal>

<Global Enumeration Defn>                 ::= Start Enum
                                              <Enumeration Identification>
                                              [ <Enumeration Type> ]
                                              <Enumeration Body>
                                              End Enum
```

```
<Enumeration Identification>            ::= Name '=' <Enumeration Name>

<Enumeration Name> ::= <MIF Literal>

<Enumeration Type> ::= Type '=' Int[eger]

<Enumeration Body> ::= <Enumeration Body> <Enum Statement> |
                                            <Enum Statement>

<Enum Statement>   ::= <MIF Integer> '=' <Enum Symbol Name>

<Enum Symbol Name> ::= <MIF Literal>

<Group Definition> ::= Start Group
                                            <Group Identification>
                                            <Group Body>
                                            End Group
<Group Identification>                  ::= <Group Name Statement>
                                            <Class Statement>
                                            [ <ID Statement> ]
                                          (Note: These statements may be in any order.
                                          If <Id Statement> is omitted, the group is a
                                          template definition.)

<Group Name Statement>                  ::= Name '=' <Group Name>

<Group Name>                            ::= <MIF Literal>

<Class Statement>  ::= Class '=' <Class String>

<Class String>                          ::= <MIF Literal>

<ID Statement>                          ::= ID '=' <MIF ID>

<Group Body>                            ::= [ <Description> ]
                                            [ <Key Statement> ]
                                            [ <Pragma Statement> ]
                                            { <Attribute Definition> }*
                                          (Note: These statements may be in any order.  If
                    this is a template definition, <Key Statement>
                                          is required.)

<Key Statement>    ::= Key '=' <Key List>

<Key List>                              ::= <Key List> , <Key> |
                                            <Key>

<Key>                                   ::= <Attribute ID>

<Pragma Statement> ::= Pragma '=' <Pragma String>

<Pragma String>    ::= <MIF Literal>

<Attribute ID>                          ::= <MIF ID>

<Table Definition> ::= Start Table
                                            <Table Identification>
                                            <Table Body>
                                            End Table

<Table Identification>                  ::= <Table Name Statement>
                                            <Class Statement>
                                            <ID Statement>
                                          (Note: These statements may be in any order.)

<Table Name Statement>                  ::= Name '=' <Table Name>

<Table Name>                            ::= <MIF Literal>

<Table Body>                            ::= <Table Body> <Table Row> |
                                            Table Row>

<Table Row>                             ::= '{' <Table Row List> '}'
```

```
<Table Row List>   ::= <Table Row List> , [ <Table Item> ] |
                                             [ <Table Item> ]

<Table Item>                                 ::= <Constant Expression>

<Constant Expression>                        ::= <Enum Symbol Name> |
                                                 '*' <Instrumentation Symbolic Name> |
                                                 <MIF Counter> | <MIF Counter64> |
                                                 <MIF Date> |
                                                 <MIF Gauge> |
                                                 <MIF OctetString> |
                                                 <MIF DisplayString> |
                                                 <MIF Integer> | <MIF Integer64>

<Attribute Definition>                       ::= Start Attribute
                                                 <Attribute Identification>
                                                 <Attribute Body>
                                                 End Attribute

<Attribute Identification>                   ::= <Attribute Name Statement>
                                                 <ID Statement>
                                                (Note: These statements may be in any order.)

<Attribute Name Statement>                   ::= Name '=' <Attribute Name>

<Attribute Name>   ::= <MIF Literal>

<Attribute Body>                             ::=    [ <Description> ]
                                                 [ <Access Statement> ]
                                                 [ <Storage Statement> ]
                                                 <Type Statement>
                                                 [ <Value Statement> ]
                                                 [ <Pragma Statement> ]
                                                (Note: These statements may be in any order,
                                                but the <Value Statement> must appear
                                                after the <Type Statement>.  The <Value Statement>
                                                is optional for templates, and required otherwise.)

<Access Statement> ::= Access '=' <Access Type>

<Access Type>                                ::= Read-Only |
                                                 Read-Write |
                                                 Write-Only |

<Storage Statement>                          ::= Storage '=' <Storage Type>

<Storage Type>                               ::= Specific |
                                                 Common

<Type Statement>   ::= Type '=' <Attribute Type>

<Attribute Type>   ::= <Enumeration Name> |
                                                 <Local Enumeration Defn> |
                                                 Counter | Counter64 |
                                                 Date |
                                                 Gauge |
                                                 OctetString <String Size> |
                                                 DisplayString <String Size> |
                                                 String <String Size> |
                                                 Int[eger] | Int[eger]64

<String Size>                                ::= '(' <Unsigned Integer> ')'

<Value Statement>  ::= Value '=' <Constant Expression> |
                                                 Value '=' Unsupported
                                                 Value '=' Unknown

<Local Enumeration Defn>                     ::= Start Enum
                                                 [ <Enumeration Identification> ]
                                                 [ <Enumeration Type> ]
                                                 <Enumeration Body>
                                                 End Enum
```

```
<MIF Literal>                          ::= '"' { <Literal Char> }* '"'

<Literal Char>                         ::= <Escape Char> |
                                           <Any ISO 8859-1 Char> |
                                           <Any Unicode Char>
                                       (Note: character encoding cannot be mixed:
                                       use ISO 8859-1 or Unicode, but not both).

<Escape Char>                          ::= <Character Escape> |
                                           <Octal Escape> |
                                           <Hexadecimal Escape>

<Character Escape> ::= '\' <Literal Escape Char>

<Literal Escape Char>                  ::= '"'  | '\' | 'a' | 'b' |
                                           'f' | 'n' | 'r' | 't' | 'v' | 'x'

<Octal Escape>                         ::= '\' <Octal Digit> { <Octal Digit> }*

<Hexadecimal Escape>                   ::= '\x' <Hex Digit> { <Hex Digit> }*

<MIF ID>                               ::= <Unsigned Integer (Non-Zero)>

<MIF Counter>                          ::= <Unsigned Integer>

<MIF Counter64>    ::= <Unsigned Integer>

<MIF Date>                             ::= <MIF Literal>
                                       (Note: The contents of the literal is in the format
                                       described in Section 2.1.4, Data types)

<MIF Gauge>                            ::= <Unsigned Integer>

<MIF OctetString>  ::= <MIF Literal>

<MIF DisplayString>                    ::= <MIF Literal>

<MIF Integer>                          ::= <Integer>

<MIF Integer64>    ::= <Integer>

<Integer>                              ::= <Decimal Integer> |
                                           <Octal Integer> |
                                           <Hexadecimal Integer>

<Decimal Integer>  ::= [ <Sign> ] <Decimal Digit> { <Decimal Digit> }*

<Octal Integer>    ::= '0' <Octal Digit> { <Octal Digit> }*

<Hexadecimal Integer>                  ::=  '0x' <Hex Digit> { <Hex Digit> }* |
                                           '0X' <Hex Digit> { <Hex Digit> }*

<Sign>                                 ::= '+' | '-'

<Unsigned Integer> ::= <Decimal Digit> { <Decimal Digit> }*
                     | <Octal Integer> | <Hexadecimal Integer>

<Octal Digit>                          ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7'

<Decimal Digit>    ::= <Octal Digit> | '8' | '9'

<Hex Digit>                            ::= <Decimal Digit> | 'A' | 'B' | 'C' | 'D' |
                                           'E' | 'F' | 'a' | 'b' | 'c' | 'd' | 'e' | 'f'

<Any ISO 8859-1 Char>                  "From ISO 8859-1 First Edition 1987-02-15
                                       Reference number ISO 8859-1: 1987 (E)"

<Any Unicode Char> "From Unicode 1.1 specification"
```

2.3 SAMPLE MIF

```
//
//          Sample MIF for the fictional ACS-100
//          mfg. by Any Computer System, Inc.
//

Start Component

            Name = "Any Computer System, Model 100"
            Description = "This component represents the base configuration"
                          "of a system manufactured by Any Computer, Inc."
                          "Three groups are included:"
                          "the ComponentID Group, "
                          "the Service Group, and "
                          "the System Chassis Group."
            Start Path
                  Name = "Chassis Group Code"
                  Dos = "C:\\any\\dos\\chassis.ovl"
                  Win16 = "C:\\any\\win3x\\chassis.dll"
            End Path

//
// Component ID Group
//
//          this is the REQUIRED group containing the
//          REQUIRED attributes for all components.
//

Start Group
            Name = "ComponentID"
            Id = 1
            Class = "DMTF|ComponentID|001"
// this group is DMTF sanctioned
            Description = "This group defines attributes common to all"
                          " components.  This group is required."

Start Attribute
            Name = "Manufacturer"
            Id = 1
            Access =  Read-Only
            Storage = Common
            Type  =  String(64)
            Value  =  "Any Computer System, Inc."
End Attribute

Start Attribute
            Name = "Product"
            Id = 2
            Access =  Read-Only
            Storage = Common
            Type  =  String(64)
            Value  =  "ACS-100"
End Attribute

Start Attribute
            Name = "Version"
            Id = 3
```

```
                Access =  Read-Only
                Storage = Specific
                Type  =  String(64)
                Value  =  "V123"
End Attribute

Start Attribute
                Name = "Serial Number"
                Id = 4
                Access =  Read-Only
                Storage = Specific
                Type  =  String(64)
                Value  =  "1234567890abcdef"
End Attribute

Start Attribute
                Name = "Installation"
                Id = 5
                Access = Read-Only
                Storage = Specific
                Type = Date
                Description = "The time and date of the (last) install of "
                              "the component"
                Value = "19930629100000.000000-300"
End Attribute

Start Attribute
                Name = "Verify"
                Id = 6
                Access = Read-Only
                Storage = Specific
                Type = Integer
                Description = "A code that provides a level of verification "
                              "that the component is still installed and working."
                Value = unknown
End Attribute

End Group                          // DMTF|ComponentID|001

//
// Service Group
//
//              The Service Group contains information regarding the servicing of
//              this system.
//

Start Group
                Name = "Service Group"
                Id = 2
                Class = "AnyComputer|SystemGroup|001"
                Description = "The Service Group contains information"
                              " about the servicing of this system."

Start Attribute
                Name = "Service Tag No."
                Id = 1
                Access =  Read-Only
                Storage = Specific
```

```
            Type  =  String(64)
            Value  =  "1234567890abcdef"
            Description = "Serial Tag Number."
End Attribute

Start Attribute
            Name = "Warranty Start Date"
            Id = 2
            Access =  Read-Only
            Storage = Specific
            Type  =  Date
            Value  = "19930107093000.000000-300"
            Description = "The start date of the service warranty."
End Attribute

Start Attribute
            Name = "Warranty Duration"
            Id = 3
            Access =  Read-Only
            Storage = Common
            Type  =  Integer
            Value  =  24        // months of duration
            Description = "The total duration of this system's warranty"
                          " in calendar months."
End Attribute

Start Attribute
            Name = "Support Phone Number"
            Id = 4
            Access =  Read-Only
            Storage = Common
            Type  =  String(64)
            Value  = "1-800-555-1234"
            Description = "The phone number(s) for support for this system."
End Attribute

Start Attribute
            Name = "Asset Number"
            Id = 5
            Access =  Read-Only
            Storage = Specific
            Type  =  String(64)
            Value  =  "Big-Corp-566-98-5725"
            Description = "The asset number for this system."
End Attribute

End Group             // Service Group
```

```
//
// System Chassis Group
//
//            The System Chassis Group
//              contains a description of the chassis
//            in this system.
//

Start Group
              Name = "System Chassis Group"
              Id = 3
              Class = "AnyComputer|SystemChassis|001"
              Description = "The System Chassis Group describes the"
                            " characteristics of this systems chassis."

Start Attribute
              Name = "System Model No."
              Id = 1
              Access =  Read-Only
              Storage = Specific
              Type  =  String(32)
              Value  =  * "Chassis Group Code"
              Description = "The System Model number for this system."
End Attribute

Start Attribute
              Name = "Physical Characteristics"
              Id = 2
              Access =  Read-Only
              Storage = Common
              Type  =  String(64)
              Value  =  * "Chassis Group Code"
              Description = "The physical characteristics of this system,"
                            " such as tower vs. slim line vs. desktop."
End Attribute

Start Attribute
              Name = "Card Slot Count"
              Id = 3
              Access =  Read-Only
              Storage = Common
              Type  = Integer
              Value  =  * "Chassis Group Code"
              Description = "The total number of card slots for this system."
End Attribute

Start Attribute
              Name = "Number Of Drive Bays"
              Id = 4
              Access =  Read-Only
              Storage = Common
              Type  =  Integer
              Value  =  * "Chassis Group Code"
              Description = "The number of half-height drive bays "
                            "in this system."
End Attribute

Start Attribute
```

```
                NAME = "POWER SUPPLY WATTAGE"
                ID = 5
                ACCESS =  READ-ONLY
                STORAGE = COMMON
                TYPE  =  INTEGER
                VALUE  =  * "CHASSIS GROUP CODE"
                DESCRIPTION = "THE WATTAGE OF THIS SYSTEM'S POWER SUPPLY."
END ATTRIBUTE


START ATTRIBUTE
                NAME = "POWER SUPPLY VOLTAGE"
                ID = 6
                ACCESS =  READ-ONLY
                STORAGE = COMMON
                TYPE  =  INTEGER
                VALUE  =  * "CHASSIS GROUP CODE"
                DESCRIPTION = "THE VOLTAGE OF THIS SYSTEM'S POWER SUPPLY."
END ATTRIBUTE

END GROUP               // SYSTEM CHASSIS GROUP

END COMPONENT
```

2.4 ISO 639

The following is included for reference only. This is not the official ISO document. It is also not part of the DMI specification, but is here for reference.For detailed information refer to the technical contents of ISO 639:1988 (E/F) "Code for the representation of names of languages".

aa	Afar	ga	Irish	mg	Malagasy	sm	Samoan
ab	Abkhazian	gd	Scots Gaelic	mi	Maori	sn	Shona
af	Afrikaans	gl	Galician	mk	Macedonian	so	Somali
am	Amharic	gn	Guarani	ml	Malayalam	sq	Albanian
ar	Arabic	gu	Gujarati	mn	Mongolian	sr	Serbian
as	Assamese			mo	Moldavian	ss	Siswati
ay	Aymara	ha	Hausa	mr	Marathi	st	Sesotho
az	Azerbaijani	hi	Hindi	ms	Malay	su	Sundanese
		hr	Croatian	mt	Maltese	sv	Swedish
ba	Bashkir	hu	Hungarian	my	Burmese	sw	Swahili
be	Byelorussian	hy	Armenian				
bg	Bulgarian			na	Nauru	ta	Tamil
bh	Bihari	ia	Interlingua	ne	Nepali	te	Tegulu
bi	Bislama	ie	Interlingue	nl	Dutch	tg	Tajik
bn	Bengali; Bangla	ik	Inupiak	no	Norwegian	th	Thai
bo	Tibetan	in	Indonesian			ti	Tigrinya
br	Breton	is	Icelandic	oc	Occitan	tk	Turkmen
		it	Italian	om	(Afan) Oromo	tl	Tagalog
ca	Catalan	iw	Hebrew	or	Oriya	tn	Setswana
co	Corsican					to	Tonga
cs	Czech	ja	Japanese	pa	Punjabi	tr	Turkish
cy	Welsh	ji	Yiddish	pl	Polish	ts	Tsonga
		jw	Javanese	ps	Pashto, Pushto	tt	Tatar
da	Danish			pt	Portuguese	tw	Twi
de	German	ka	Georgian				
dz	Bhutani	kk	Kazakh	qu	Quechua	uk	Ukrainian
		kl	Greenlandic			ur	Urdu
el	Greek	km	Cambodian	rm	Rhaeto-Romance	uz	Uzbek
en	English	kn	Kannada	rn	Kirundi		
eo	Esperanto	ko	Korean	ro	Romanian	vi	Vietnamese
es	Spanish	ks	Kashmiri	ru	Russian	vo	Volapuk
et	Estonian	ku	Kurdish	rw	Kinyarwanda		
eu	Basque	ky	Kirghiz			wo	Wolof
				sa	Sanskrit		
fa	Persian	la	Latin	sd	Sindhi	xh	Xhosa
fi	Finnish	ln	Lingala	sg	Sangro		
fj	Fiji	lo	Laothian	sh	Serbo-Croatian	yo	Yoruba
fo	Faeroese	lt	Lithuanian	si	Singhalese		
fr	French	lv	Latvian, Lettish	sk	Slovak	zh	Chinese
fy	Frisian			sl	Slovenian	zu	Zulu

2.5 ISO 3166

The following is included for reference only. This is not the official ISO document. It is also not part of the DMI specification, but is here for reference. Students of political science will note that some of these entries are out of date. For detailed information refer to the technical contents of ISO 3166:1988 (E/F) "Code for the representation of names of territory". ISO 3166 defines 2-letter codes, 3-letter codes and numeric codes. The DMI uses only the 2-letter codes.

Afghanistan	AF	Chile	CH	Greenland	GL
Albania	AL	China	CN	Grenada	GD
Algeria	DZ	Christmas Island	CX	Gudeloupe	GP
American Samoa	AS	Cocos (Keeling) Islands	CC	Guam	GU
Andorra	AD	Colombia	CO	Guatemala	GT
Angola	AO	Comoros	KM	Guinea	GN
Anguilla	AI	Congo	CG	Guinea-Bissau	GW
Antarctica	AQ	Cook Islands	CK	Guyana	GY
Antigua & Barbuda	AG	Costa Rica	CR		
Argentina	AR	Cote D'Ivoire	CI	Haiti	HT
Aruba	AW	Cuba	CU	Heard & McDonald I.	HM
Australia	AU	Cyprus	CY	Honduras	HN
Austria	AT	Czechoslovakia	CS	Hong Kong	HK
				Hungary	HU
Bahamas	BS	Denmark	DK		
Bahrain	BH	Djibouti	DJ	Iceland	IS
Bangladesh	BD	Dominica	DM	India	IN
Barbados	BB	Dominican Republic	DO	Indonesia	ID
Belgium	BE			Iran (Islamic Republic)	IR
Belize	BZ	East Timor	TP	Iraq	IQ
Benin	BJ	Ecuador	EC	Ireland	IE
Bermuda	BM	Egypt	EG	Israel	IL
Bhutan	BT	El Salvador	SV	Italy	IT
Bolivia	BO	Equatorial Guinea	GQ		
Botswana	BW	Ethiopia	ET	Jamaica	JM
Bouvet Island	BV			Japan	JP
Brazil	BR	Falkland I (Malvinas)	FK	Jordan	JO
British Indian O. Terr.	IO	Faroe I.	FO		
Brunei Darussalam	BN	Fiji	FJ	Kampuchea, Democratic	KH
Bulgaria	BG	Finland	FI	Kenya	KE
Burkina Faso	BF	France	FR	Kiribati	KI
Burma	BU	French Guiana	GF	Korea,Dem. People's Rep	KP
Burundi	BI	French Polynesia	PF	Korea, Rep. of	KR
Byelorussian SSR	BY	French Southern Terr.	TF	Kuwait	KW
Cameroon	CM	Gabon	GA	Lao People's Dem. Rep.	LA
Canada	CA	Gambia	GM	Lebanon	LB
Cape Verde	CV	Germany	DE	Lesotho	LS
Cayman Islands	KY	Ghana	GH	Liberia	LR
Central African Rep.	CF	Gibraltar	GI	Libyan Arab Jamahiriya	LY
Chad	TD	Greece	GR	Liechtenstein	LI
Luxembourg	LU	Philippines	PH	Tunisia	TN

		Pitcairn Island	PN	Turkey	TR
Macau	MO	Poland	PL	Turks and Caicos Isl.	TC
Madagascar	MG	Portugal	PT	Tuvalu	TV
Malawi	MW	Puerto Rico	PR		
Malaysia	MY			Uganda	UG
Maldives	MV	Qatar	QA	Ukranian SSR	UA
Mali	ML			United Arab Emirates	AE
Malta	MT	Reunion	RE	United Kingdom	GB
Marshall Islands	MH	Romania	RO	United States	US
Martinique	MQ	Rwanda	RW	US Minor Outlying I.	UM
Mauritania	MR			Uruguay	UY
Mauritius	MU	St. Helena	SH	USSR	SU
Mexico	MX	Saint Kitts and Nevis	KN		
Micronesia	FM	Saint Lucia	LC	Vanuatu	VU
Monaco	MC	St. Pierre & Miquelon	PM	Vatican City State	VA
Mongolia	MN	St.Vincent & Grenadines	VC	Venezuela	VE
Montserrat	MS	Samoa	WS	Viet Nam	VN
Morocco	MA	San Marino	SM	Virgin Islands (British)	VG
Mozambique	MZ	Sao Tome and Principe	ST	Virgin Islands (US)	VI
		Saudia Arabia	SA		
Namibia	NA	Senegal	SN	Wallis and Futuna Isl.	WF
Nauru	NR	Seychelles	SC	Western Sahara	EH
Nepal	NP	Sierra Leones	SL		
Netherlands	NL	Singapore	SG	Yemen	YE
Netherlands Antilles	AN	Solomon Islands	SB	Yemen, Democratic	YD
Neutral Zone	NT	Somalia	SO	Yugoslavia	YU
New Caledonia	NC	South Africa	ZA		
New Zealand	NZ	Spain	ES	Zaire	ZR
Nicaragua	NI	Sri Lanka	LK	Zambia	ZM
Niger	NE	Sudan	SD	Zimbabwe	ZW
Nigeria	NG	Suriname	SR		
Niue	NU	Svalbard & Jan Mayen I.	SJ		
Norfolk Island	NF	Swaziland	SZ		
Northern Mariana I.	MP	Sweden	SE		
Norway	NO	Switzerland	CH		
		Syrian Arab Republic	SY		
Oman	OM				
		Taiwan	TW		
Pakistan	PK	Tanzania, United Rep.	TZ		
Palau	PW	Thailand	TH		
Panama	PA	Togo	TG		
Papua New Guinea	PG	Tokelau	TK		
Paraguay	PY	Tonga	TO		
Peru	PE	Trinidad and Tobago	TT		

3. STANDARD GROUPS

This section describes the three important classes of standard groups for this version of the DMI. They are the ComponentID group, the Event Groups, and the DMI Service Provider Groups. The ComponentID group is one that must be implemented by all DMI components. The Event groups include a template group used to describe the format of event data for standard events. In addition an Event State group is defined to hold the current state of state-based events An event example is provided at the end of this section. The Service Provider standard groups are required to be implemented by all DMI Service Provider implementations.

3.1 COMPONENT STANDARD GROUPS

3.1.1 The ComponentID group

Every MIF file must contain a standard group with ID 1. This group offers base-level identification of the component and represents the minimum amount of information that a component vendor should provide (when meaningful). An attribute that is not supported or that has no meaning for a given component should give the keyword **unsupported** or **unknown** as its value.

The ComponentID class string is "DMTF|ComponentID|001".

The six named attributes in the group are: "Manufacturer", "Product", “Version”, "Serial Number", "Installation", and "Verify". Their definitions are:

3.1.1.1 MANUFACTURER

```
Name = "Manufacturer"
ID = 1
Description = " The organization that produced this component”
Access = Read-Only
Storage = Common
Type = String(64)
```

3.1.1.2 PRODUCT

```
Name = "Product"
ID = 2
Description = “The name of this component or product”
Access = Read-Only
Storage = Common
Type = String(64)
```

3.1.1.3 VERSION

```
Name = “Version”
ID = 3
Description = “The version string for this component”
Access = Read-Only
Storage = Specific
Type = String(64)
```

3.1.1.4 SERIAL NUMBER

```
Name = "Serial Number"
ID = 4
Description = “The serial number for this component”
Access = Read-Only
Storage = Specific
Type = String(64)
```

3.1.1.5 INSTALLATION

```
Name = "Installation"
ID = 5
Description = “The time and date of the last install of the component on this”
     “system”
Access = Read-Only
Storage = Specific
Type = date
```

3.1.1.6 VERIFY

```
Name = "Verify"
ID = 6
Description = “The verification level for this component”
Access = Read-Only
Storage = common
Type = integer
```

Asking for the value of the “Verify” attribute causes the component instrumentation to perform checks to verify that the component is still in the system and working properly. It should return one of the following values:

VALUE	MEANI IG
0	an error occurred; check status code
1	component does not exist
2	verify not supported
3	RESERVED
4	component exists, functionality untested
5	component exists, functionality unknown
6	component exists, functionality no good
7	component exists, functionality good

3.2 EVENT STANDARD GROUPS

This section describes a model for producing standard DMI events and also provides mechanisms that vendors may use to extend standard events to produce proprietary event types.

An *Event* is the manifestation of a change of state, or the occurrence of condition of interest with a hardware or software device. The generation of an Event causes the DMI Service Provider to directly or indirectly process it. An *Indication* is a notification of an Event to an event consumer. Indications include Event notifications as well as notifications of changes in the DMI Service Provider's database, e.g. notification that a Component or a Group has been added to or deleted from the database, that a Component has been installed or uninstalled.

An *Event Generator* is hardware or software device that has undergone a change in state or in which a certain condition of interest has occurred. An *Event Consumer* is an entity that is interested in receiving notification of the occurrence of an Event of interest. This change of state or condition will directly or indirectly cause a new event to be processed by the DMI Service Provider which then produces and delivers an Indication data structure to event consumers that have expressed their interest in receiving Indications. An *Event Reporter* is a software entity that causes a new DMI event to be processed by the Service Provider, either on its own behalf (in which case it is also an Event Generator), or on behalf of another Event Generator entity. Events are “reported” by calling the Service Provider entry point *DmiOriginateEvent*.[2]

Event consumers must express their interest in receiving event notifications through a *subscription* mechanism described later in this chapter. Upon the reporting of an Event, the DMI Service Provider produces and delivers a data structure (an Indication) containing data describing the Event to all event consumers that have subscribed to receive Indications.

Event consumers could, of course, be remote relative to the DMI Service Provider. In this case it is desireable not to propagate all event notifications to the remote site across the intervening communication medium. This implies the need for a *filtering* mechanism for event notifications. Such a filtering mechanism is specified later in this chapter. The DMI Service Provider matches each event against filters provided by a remote consumer to determine whether or not a specific Indication should be delivered to that remote consumer.

When an Indication is delivered to an event consumer, the event data appear to the consumer exactly as though the consumer had done a DMI Get operation to a functional group; we say that the Event data appear as though they were the result of an "unsolicited Get". Naturally, therefore, the event data need to be formatted as a DMI group. To describe this format we introduce the notion of a *Event Generation Group* which is really only a template. The syntactic definition of this group appears very much like that of normal groups. However, its role is solely that of a template to define the format of event data. Consequently, we distringuish this special format-defining group through a special form of class string.

When a consumer receives an Indication the data structure contains a *DmiMultiRowData* structure within it. Each *DmiMultiRowData* structure is composed of possibly multiple *DmiRowData* structures. This chapter describes the format of the first two *DmiRowData* structures for standard Indications. (See Section 5.3 for definitions of these data structures)

Some key aspects of the event model described in this chapter are:

- **An Event Generation Group**

As described above, this group is a template for, and defines the “format” of standard events. By interpreting the delivered Indication data according to this format, the management application can display a localized[3] description of the cause (and possibly solution) of the event.

This chapter also describes a mechanism whereby a vendor can extend, in a proprietary manner, the set of events described by a standard event generation group.

- **An Event State Group**

The Event State Group defines a table, each of whose rows represents the state of a state-based event, within the Component where the Event State Group is instrumented. A state-based event can occur when the state of the event generating device changes. Most typically, a state-based event might be generated when (a) a device encounters a problem and enters a problem state, or, (b) when the problem is cleared and the device re-enters its normal operating state. An instance of the Event State Group must be included in every Component that generates state-based events.

[2] or an analogous native entry point in OSes that do not implement the CI
[3] i.e. translated into the appropriate language.

3.2.1 Requirements

3.2.1.1 MIF REQUIREMENTS

Each group in the MIF that represents Event Generator(s) must have a corresponding Event Generation Group (See Section 3.2.2). It is recommended that each Event Generation group immediately follow the referenced group, and that the Event Generation group's ID value is the numeric successor of the referenced group's ID value.

Additionally, if the Event Generation group is capable of generating state-based events (which is the usual case), then there must be an instance of the Event State group defined in the Component that contains the Event Generation group.

3.2.1.2 EVENT REPORTER REQUIREMENTS

For events that may be associated with a particular instance of a group (a row in a table), Event Reporters must provide instance-specific data (i.e. a keylist) in the second *DmiRowData* structure within the Indication data structure.

Software entities that are not registered as components with the DMI Service Provider may act as Event Reporters by calling the *DmiOriginateEvent* entry point in the Component Interface (CI), or its equivalent in the operating system environment in question. This would typically occur in situations where that software entity is reporting a "synthetic event"; an event that is generated based on a composite analysis of various elements of state in the managed machine. In such a case, the reported Component ID field in the Indication data structure must be zero. Likewise, the reported Class String of the event generating group must be a null string.

3.2.2 Event Generation Group

This section describes the "skeleton" or template for a group that is used for event generation. The Event Generation Group definition is in a template form and is not a true group definition. The reason for this is that the event definition contains elements that must be tailored for the group representing the entity(s) actually causing the event(s).

Structure of event data

The event data received by an event consumer will consist of one or more *DmiRowData* structures (i.e. a *DmiMultiRowData* structure). For standard events the following conditions apply to these *DmiRowData* structures:

- The first *DmiRowData* structure contains a row whose format is identical to that of the Event Generation Group defined below in this section.
- The second *DmiRowData* structure contains a keylist in the case that the event generating group is a tabular group. This keylist selects the precise row of the tabular group that was the Event Generator (e.g. the event generating Processor in a table of Processors).
- The third *DmiRowData* structure is reserved for carrying addressing information describing the node that originated the event in the case that the event is (multiply) forwarded to its eventual destination across a communication medium.
- Fourth and subsequent *DmiRowData* structures, if they exist, may contain any additional (proprietary) information that is required to further elaborate on the event.

Vendor proprietary events

Vendor proprietary events need not adhere to these conditions, but then their event data will not be recognized or processed by all DMI management applications. A mechanism using an extended class string format is described below for those vendors wishing to provide proprietary indications while staying within the above conditions.

Template definition and class string

Attribute definitions within a non-tabular group must have a value statement. The attribute values in template group definition below are arbitrary; they are provided only for syntactic completeness, so that they will not cause errors when processed by MIF parsers and processors. In practice, Management Applications will not access these values defined in the template — rather, Management Applications will use values directly from the Indication data structure that is delivered to a consumer of Indications. (An exception to this rule is Attribute 5, the *Associated Group* Attribute. The value of this attribute identifies the Event Generator group and therefore must be a valid attribute value even within the template.) The template group definition is used by Management Applications to associate values in the Indication data structure with enumeration display strings. The definition of the event generation group will start as follows:

```
Name = "Event Generation"
Class = "EventGeneration|<Specific name>|002"
```

```
ID =
Key = 5
```

Note here that the version number in the class string for the Event Generation template refers to the version of the template.

Each event generation group will have a unique class string in which the <Specific name> field above is constructed according to the following format[4]:

```
<defining-body> <delim> <specific-name-of-assoc-group>
or
<defining-body> <delim> <specific-name-of-assoc-group> <delim> <proprietary-
                        extension>
```

where <delim> = ^^ (i.e. two caret characters in sequence)

It is suggested that the *proprietary-extension* field contain additional characters that make the field unique. To accomplish this, component vendors who wish to include additional event types for a standard event generation group should augment the *proprietary-extension* field with additional descriptive text. In particular, the full, registered name of the corporate entity of the vendor should be used to ensure uniqueness of the *specific-name* field of the event generation group.

For example, if the DMTF Server Working Committee wished to define an Event Generation group for the *UPS Battery* standard group, they might choose:

```
"EventGeneration|DMTF^^UPS Battery|002"
```

as its class string. A UPS vendor, named say "Excellent Power Systems, Inc." wishing to define an additional proprietary event condition for their UPS batteries might choose, for example:

```
"EventGeneration|DMTF^^UPS Battery^^Low Electrolyte"
" Excellent Power Systems, Inc.|002"
```

as the class string.

Of course, vendors may choose to define entirely proprietary sets of events by using the full registered name of their corporate entity in the *defining-body* portion of the class string. If the format of the EventGeneration template is maintained in the first, second and third RowData structures of the Indication data, then these proprietary events could still be manipulated in simple ways by any DMI management application. However, their full semantics would only be known to the vendors' own proprietary management applications.

It is suggested that when defining multiple Event Generation templates for a single Event Generator group, that they all appear immediately following the associated group in the MIF, and that they have sequential group IDs.

The value of this group's ID may be any unused ID. The key is used by Management Applications to discover the associated group. See "Associated Group" in Section 3.2.2.2.5.

3.2.2.1 COMMON DEFINITIONS

```
Start Enum
Name = "BOOL"
     0 = "False"
     1 = "True"
End Enum
```

3.2.2.2 DEFINITIONS OF REQUIRED ATTRIBUTES

The following attributes **must** be included in the definition of a standard Event Generation group. See Section 3.2.3.2.

---

[4]Rationale:

A. The use of another type of delimiter in the class string for the EventGeneration template, over and above the '|' character, is required to
   1. distinguish different defining bodies (e.g. user groups such as OURS),
   2. disambiguate the cases "StdGroup", "StdGroup Capabilities", and "StdGroup MyTemplate" where the first two are standard group names and the third one is a proprietary event extension to the "StdGroup" event generator. In other words there is no way to tell that "StdGroup MyTemplate" is proprietary and "StdGroup Capabilities" is standard unless the MA has an up-to-date list of all standard class names.
   3. provide clarity and readability

B. A delimiter composed of an unlikely string of multiple characters is specified so that the use of the individual characters is still retained. Also, current parsers will not break.

3.2.2.2.1 Event Type

The “reason” that the event occurred. For example, a printer may be able to generate *JAM* events.

```
Name = "Event Type"
ID = 1
Description = "The type of event that has occurred."
Type = <Enum>
Access = Read-Only
Storage = Specific
Value = unknown
```

Note that the enumeration is not defined here. Each Event Generation group will have a unique definition for this attribute.

3.2.2.2.2 Event Severity

The event severity describes the type of event. *Monitor* and *Information* events are not associated with the state of the entity generating the event and are used to convey information. *OK*, *Non-Critical*, *Critical*, and *Non-Recoverable* events are state-based and represent successively more serious abnormal conditions.

Monitor events are used by transaction-oriented event generators. *Monitor* events are periodic in nature and are expected to be encountered by event consumers. An example of a *Monitor* event would a lock/unlock operation from a database server.

Information events are used to indicate a non-problematic change that is non-periodic in nature. An example of an *Information* event would be a paper size change in a paper tray of a printer.

OK events inform the event consumer that the entity generating the event has entered the OK or “normal” state. On initialization a device may generate this event. State-based generators will produce this event after a *Non-Critical*, *Critical*, or *Non-Recoverable* error state has “cleared.”

Non-Critical events convey a problem that needs to be corrected. However, they do not imply a specific time period within which corrective action(s) need to be taken. For example, a printer that had two paper trays may generate a *Non-Critical* event when one of them runs out of paper.

A *Critical* event is more serious. These problems need to be corrected usually within a specific time period whose duration is governed by the device type and/or the particular problem situation. For example, if a printer has only one paper tray, and that tray runs out of paper, printing cannot continue. In this scenario, the printer may generate a *Critical* event. A time period may be associated with this event after which, if the paper tray is not replenished, the print job might be discarded.

A *Non-Recoverable* event is the most serious. Not only must it be corrected immediately for an operation to proceed, but the cause of the failure itself is severe. Failures in devices that can only be corrected by cycling the power, or performing an off-line repair operation are *Non-Recoverable* events.

The contents of the event state field within the rows of the Event State group associated with the Component, in which the Event Generator group is located, will contain one of the following four Severities at any time: *OK*, *Non-Critical*, *Critical*, *Non-Recoverable*.

```
Name = "Event Severity"
ID = 2
Description = "The severity of this event."
Type =        Start Enum
              0x001 = "Monitor"
              0x002 = "Information"
              0x004 = "OK"
              0x008 = "Non-Critical"
              0x010 = "Critical"
              0x020 = "Non-Recoverable"
    End Enum
Access = Read-Only
Storage = Specific
Value = unknown
```

The enumeration defined in this attribute must not be changed. This is to allow this same enumeration to be used to filter events.

3.2.2.2.3 Event Is State-Based

Event generators may be state-based or non state-based. State-based generators generate an event anytime the device changes state. Furthermore, for each non-normal event generated, an *OK* event will be generated when that condition clears. If the printer runs out of paper in bin one (and generates a *Non-Critical* event), and develops a jam in the output

path (generating a *Critical* event), then that printer will generate an *OK* event for **each** of those events when they are corrected.

It is presumed that state-based event generators generate no more than one event of any given event type for each relevant state transition.

A non state-based generator will issue an event for each condition of interest that develops, but does not issue corresponding *OK* events as above.

This attribute takes the value TRUE if the Event being reported is state-based. Otherwise, it takes the value FALSE.

```
Name = " Event Is State Based"
ID = 3
Description = "The value of this attribute determines whether the Event being”
     “reported is a state-based Event or not. If the value of this attribute”
                “is TRUE then the Event is state-based. Else the Event is not state-
     “based."
Type = "BOOL"
Access = Read-Only
Storage = Common
Value = unknown
```

3.2.2.2.4 Event State Key

This attribute has meaning if and only if the Event being reported is state-based, i.e. the value of the attribute above (Event Is State-Based) is TRUE (see Section 3.2.2.2.3). This attribute holds a single integer key that identifies a row in the Event State group associated with the Component within which the Event Generator group is located. The Current State attribute within that row holds the value of the current state of the Event. The contents of the Current State attribute are one of four enumerated severity levels (not including *Monitor* and *Information*)

```
Name = "Event State Key"
ID = 4
Description = "This attribute holds the key identifying a row of the Event State group”
              “within the Component in which the event generator group is located.  The”
              “Current State attribute within the row contains the current state of this”
              “state-based event.  The current state can be one of the four severities:  “
              “OK, Non-Critical, Critical, and Non-Recoverable."
Type = Integer
Access = Read-Only
Storage = Specific
Value = unknown
```

3.2.2.2.5 Associated Group

This attribute contains the value of the class string of the associated group i.e. the Event Generator group. This is a keyed attribute. A Management Application that discovers an Event Generation template group can find the associated group by using a DmiListComponentsByClass command with a class filter of “EventGeneration||” and a keylist with this attribute’s value.

```
Name = "Associated Group"
ID = 5
Description = "The class string of the group that is associated with the events”
              “defined in this Event Generation group."
Type = String ( <Size> )
Access = Read-Only
Storage = Common
Value = "<ClassString>"
```

The value of this attribute should be defined in the MIF. For example, if this Event Generation group defines events for the *Processor* group defined in the Systems Standard Groups Definition, V1.0, then this value would be “DMTF|Processor|003”.

3.2.2.2.6 Event System

The event system attribute indicates the functional system of the product that caused the event. For example a printer might define *Engine*, *Feeder*, and *Sorter* as functional systems of the printer. A simple management application could use the values of the Event System and Event subsystem attributes (see below) to construct a simple message describing the event.

```
Name = "Event System"
ID = 6
Description = "The major functional aspect of the product causing the fault."
Type = <Enum>
Access = Read-Only
```

```
Storage = Specific
Value = unknown
```

Note that the enumeration is not defined here. Each Event Generation Template will have a unique definition for this attribute.

3.2.2.2.7 Event Subsystem

The event subsystem attribute indicates the functional subsystem of the product that caused the event. For example a printer might define *BIN1* and *BIN2* as functional subsystems of the printer. A simple management application could use the values of the Event System (see above) and Event subsystem attributes to construct a simple message describing the event.

```
Name = "Event Subsystem"
ID = 7
Description = "The minor functional aspect of the product causing the fault."
Type = <Enum>
Access = Read-Only
Storage = Specific
Value = unknown
```

Note that the enumeration is not defined here. Each Event Generation Template will have a unique definition for this attribute.

3.2.2.3 DEFINITIONS OF OPTIONAL ATTRIBUTES

The following attributes may be included or excluded from the definition of standard Event Generation Groups. See Section 3.2.2.

3.2.2.3.1 Event Solution

The event solution attribute describes a solution to the problem that caused the event. The vendor of a product generating this event may choose to provide a string here that describes what the user of the Management Application must do to correct the problem. This string may also specify a time period within which action must be taken in the case that a *Critical* event is being reported.

```
Name = "Event Solution"
ID = 8
Description = "A solution to the problem that caused the event."
Type = <Enum>
Access = Read-Only
Storage = Specific
Value = unknown
```

Note that the enumeration is not defined here. Each Event Generation Template will have a unique definition for this attribute. The set of possible solution strings are provided here as an enumeration so that they may be easily localized to the desired language of the end-user of the Management Application.

3.2.2.3.2 Instance Data Present

This attribute is used to inform the Management Application that the second *DmiRowData* data structure within the Indication data structure contains instance-specific data...For example, if an event template were constructed to support the *Processor* group from the Systems Standard Groups Definition, then it would be desirable if an event not only described a particular processor fault, but also which processor in the table was the one that caused the failure.

```
Name = "Instance Data Present"
ID = 9
Description = "Indicates whether the second event block contains instance-specific data."
Type = "BOOL"
Access = Read-Only
Storage = Specific
Value = unknown
```

3.2.2.3.3 Vendor Specific Message

The following two attributes allows the product supplier to define a “private” interface between the producer and the consumer of an event. Producers of events are usually the instrumentation code associated with a product, but may in fact be any active task. Consumers are Management Applications that have registered with the DMI Service Provider to receive indications. Manufacturers who develop products that encompass both producers and consumers may find that these attributes provide an efficient, easy-to-use method of passing arbitrary information. In particular, they may use these attributes to fold existing proprietary solutions into the DMI Indications paradigm.

This attribute is used to pass displayable string data.

```
Name = "Event Message"
ID = 10
Description = "Auxiliary information related to the event."
Type = String(<Size>)
Access = Read-Only
Storage = Specific
Value = unknown
```

Note that the string definition has no maximum size associated with it. Implementors of this template may choose whatever maximum size is convenient for the set of strings defined for this attribute.

3.2.2.3.4 Vendor Specific Data

This attribute is used to pass arbitrary data.

```
Name = "Vendor Specific Data"
ID = 11
Description = "Auxiliary information related to the event."
Type = OctetString(<Size>)
Access = Read-Only
Storage = Specific
Value = unknown
```

Note that the octetstring definition has no maximum size associated with it. Implementors of this template may choose whatever maximum size is appropriate for this attribute.

3.2.3 Event State Group

The Event State group is a table keyed with a single integer which is a unique identifier for each row of the table. Each row of this table holds information about a unique single event type that is generated from a given Event Generation group within the event generating Component. The Event State group only carries the current state of state-based events within the Component.

NOTE: Unlike the event generation template defined in Section 3.2.2, this is a true group definition with the usual form of Class String.

In theory there is one event state table per location within a component which generates events, and it holds the current state of the events generated at that location. However, for simplicity, the Event State Group combines these theoretical tables into one single table in a Component, wherein each entry holds the state of one event type and "points back" to the event generation group at the event generating location within the Component.

For each row of this keyed group the *Event Generation Group* attribute carries the ID of the event generation group that defines the event type represented by the row. Management applications may scan for all state based events within a system by using a class filter of "|Event State|" to discover instances of this group. Then for each instance of this group the application may scan the rows of this group to discover state-based events.

A vendor desiring to maintain current state for proprietary state-based events may simply include additional rows within this group that "point" to the vendor's proprietary event generation group. This is done by assigning the class string of their proprietary event generation group (see Section 3.2.2) as the value of the *Event Generation Group* attribute in those additional rows.

```
Name = "Event State"
Class = "DMTF|Event State|001"
ID =
Key = 1
```

3.2.3.1 EVENT INDEX

This is a unique index for rows of this table.

```
Name = "Event Index"
ID = 1
Description = "A unique index into the Event State table"
Type = Integer
Access = Read-Only
Storage = Common
Value = unknown
```

3.2.3.2 EVENT GENERATION GROUP

This attribute contains the class string of the Event Generation group within this Component that described the Indication format for the related Event. The Component ID of the component from which the Event arose is reported in the header of the Indication data structure that is received by the Event Consumer(s).

```
Name = "Event Generation Group Class"
ID = 2
Description = "The Class String of the event generator group within the generating”
              “Component"
Type = String (256)
Access = Read-Only
Storage = Common
Value = unknown
```

3.2.3.3 EVENT TYPE

This attribute contains the type of the Event that was generated. The value of this attribute is the integer value of one of the enumerated items in the Event Type attribute in the associated Event Generation group (see Section 3.2.2.2.1). The Event Generation group in question can be identified by the attribute defined immediately above (see Section 3.2.3.2)

```
Name = "Event Type"
ID = 3
Description = "Integer value that identifies one of the Event types enumerated”
              “in the associated Event Generation group"
Type = Integer
Storage = Common
Value = unknown
```

3.2.3.4 CURRENT STATE

This attribute contains the current state (i.e. severity) of the specific event type represented by this row of the group.

```
Name = "Current State"
ID = 4
Description = "The current state of the Event type identified by Event Type”
              “attribute in this row."
Type = Start ENUM
     0x0004 = "OK"
     0x0008 = "Non-Critical"
     0x0010 = "Critical"
     0x0020 = "Non-Recoverable"
     End ENUM
Access = Read-Only
Storage = Specific
Value = "OK"
```

The enumeration defined in this attribute is a subset of the Event Severity enumeration defined in the Event Generation group. It is kept aligned with that enumeration because it reflects the current severity of the event type within the event generating component.

3.2.3.5 ASSOCIATED GROUP KEYS

This attribute exists to identify an instance of the Associated Group that may generate the state-based event in question.

For instance, consider that the Associated Group is the *Disks* group in the Systems Standard Groups Definition. This is a table group keyed by a pair of keys. The first key is an integer in an Enum called Storage Type, the second key is an integer index within a storage type. If a disk in the Disks table generates a state-based event (e.g. "disk failure") then its related entry in the Event State table must be able to identify the specific disk that was the source of the event... not just that some disk sourced the event.

In the case of an arbitrarily keyed Associated Group there could be a number of different keys each of a different type. However, we restrict the possible keys here to be Integers only (this includes simple table indexes as well as Enums. This should cover the majority of practical cases.

To represent a KeyList of integers keys we use an encoded string, the contents of which are a comma-separated list of integers without any spaces. The simple BNF for the grammar of this string is:

```
<KeyListString> ::= '"' <Integer> { ',' <Integer> }* '"'
```

where `<Integer>` is as defined in the MIF Grammar in Section 2.2.

```
Name = "Associated Group Keys"
ID = 5
Description = "A list of integer keys that identify the instance of the”
              “Associated Group that actually" "generated the state-based event.”
              “The list of integer keys are represented in the value of this”
              “attribute as a string containing a comma-separated list of"
              "integers. The management application must parse this string to”
              “obtain the list of integer keys.”
Type = String(256)
Access = Read-Only
Storage = Specific
Value = unknown
```

3.3 DMI SERVICE PROVIDER STANDARD GROUPS

When Indications are sent to remote consumers, it is desireable to limit the set of indications that are actually transmitted on the intervening communication medium. To achieve this indication consumers are required to *subscribe* for indications at each potential indication-originating node in the network. In addition, the mere act of subscribing for indications enables only the sending of notification of DMI Service Provider database changes to the consumer (e.g. "component added/deleted", "group added/deleted", etc.). If Event notifications are desired, event consumers must provide *filters* that select the specific event notifications they are interested in receiving. This section describes the mechanisms for subscription and filtering. ***DMIv2.0s*** introduces new standard groups to configure the security features, and to define security indications. These groups are defined in sections 12 and 16 respectively.

Subscription and Filter table groups

There are two groups defined for use with the Indication subscription and filtering process. Each group is instantiated as a table, where the addition or deletion of indication subscription and filter entries is handled as ADD/DELETE row operations. It is the responsibility of the DMI Service Provider to manage and use these tables. To the user of the MI interface, they will simply appear as two additional tables instantiated in the DMI Service Provider component. An important distinction is that the subscription applies to all DMI indications, while the filter applies only to that subset of indications called events. In other words, if a managing system simply adds an indication subscription entry in a managed node, it will receive all indication that are not classified as events. It will only receive the indications classified as events if it has added the appropriate filter table entry. **NOTE**: A consumer of indications must first subscribe for events and then specify filters. A consumer may have only a single subscription but may specify multiple filters.

Persistence of subscriptions

Subscriptions and Filters are intended to be persistent so that indications would continue to be delivered even if a managing system dropped off the communication medium, or was otherwise inaccessible, for some period, before returning. Likewise, subscriptions and filters are intended to be persistent over periods when the DMI Service Provider is itself not functioning. However, it is not desireable for subscriptions and filters to be so long-lived that they outlive the event consumer that specified them. To achieve this, each Indication subscription has a pair of associated timestamps, namely, an expiration warning timestamp and a expiration timestamp. These timestamps are specified by the consumer when subscribing. At the time specified by the expiration warning timestamp the DMI Service Provider sends an expiration warning indication to the *DmiSubscriptionNotice* entry point of the consumer. Likewise, at the time specified by the expiration timestamp, the DMI Service Provider sends an expiration indication to the *DmiSubscriptionNotice* entry point of the consumer. **NOTE**: When a subscription expires, the DMI Service Provider removes the row corresponding to the subscrption in the *SP Indication Subscription* table and all associated filter rows in the *SP Filter Information* table. These may be identified by matching the subscriber address fields of the subscription and the filters.

Indication retry threshold

The DMI Service Provider makes its best efforts to deliver indications despite outages of itself, the intervening communication medium, or the event consumer. If indication delivery is not possible because of such outages, it retries the delivery after waiting a reasonable period to allow the outage to clear. The maximum number of such retries is specified by the event consumer in the *Indication Failure Threshold* attribute within the *SP Indication Subscription* group defined below[5].

Indication entry points in the client

Event notifications are delivered to the event consumer at the *DmiDeliverEvent* entry point. As noted above, event notifications will not be delivered unless the consumer has specified filters for those events. There are specific individual entry points for notification of DMI Service Provider database changes (e.g. DmiComponentAdded, DmiGroupAdded, DmiComponentDeleted, DmiGroupDeleted, ... etc.). If a managing system does not wish to receive one of this latter set of indications it simply does not implement and/or publish the specific entry point. Please refer to the Interface Description Language (IDL) description of the Indication Delivery Interface for precise details of these entry points.

[5] It is expected that DMI Service Provider implementations will also choose to log at least the fact that the maximum retry threshold was exceeded. In this case the event data of the undelivered indication should also be logged. Of course, DMI Service Providers may also choose to log all events. It is expected that DMI Service Providers will use the native OS logging mechanisms and this document does not specify a separate logging mechanism.

3.3.1 SP Indication Subscription

This group will be instantiated as a table by the DMI Service Provider. It is simply a list of managing nodes that have subscribed with this managed node to receive indications. This group is used to store the information about a managing node that is required in order for the managed node to correctly forward indications. It is meant to be persistent over reboots until the time specified by the "Subscription Expiration Datestamp" attribute, defined below. The values in this group are set and modified by using the DmiAddRow(), DmiDeleteRow(), DmiGetAttribute(), and DmiSetAttribute().

Subscriber address information

Note that the set of subscriber addressing information specified includes an RPC Type and a Transport Type. This is because this version of DMI supports multiple standard RPCs, each of which is multi-transport. Thus the DMI Service Provider sending the Indication needs to know which RPC and transport must be used to reach a particular subscriber.

Single versus multiple management applications on the client node

In most cases, the managing node has running on it a single management application. This management application then needs to implement the indication delivery entry points described in the Indication Delivery Interface (see the IDL description of this interface in the appendix). The management application also publishes these indication entry points as available RPC service end points in the appropriate RPC naming services (e.g. Cell Directory Services in the case of DCE/RPC). The DMI Service Provider sending the indication then binds to these RPC service end points before calling the appropriate entry point to deliver the indication.

The situation may be slightly different in the case of a management node that is hosting multiple management applications simultaneously. There are two possibilities in this case, namely:

- Each individual management application publishes its indication entry points as RPC service end points separate and distinct from those of the other management applications on the node. In this case, each management application will have its own subscription and filter entries registered at the DMI Service Provider sending the indication.
- The managing node implements an optional "front-end" software entity that supports multiple simultaneous management applications on the managing node and insulates these management applications from the specifics of dealing with the underlying RPCs (see Section 9 "Optional MI Support Functions"). In this case, the RPC service end points are published by the front-end so that all indications, intended for the management applications it supports, are delivered to it alone. The front-end also subscribes for indications and provides filters on behalf of the multiple management applications. In other words there will be a single subscription entry and a set of filter entries corresponding solely to, and managed solely by the front-end on behalf of the management applications it supports. In this situation, when an indication is delivered to the front-end, it needs to be able to distinguish which management application is the intended final destination for the indication. To achieve this local "routing" of indications to management applications, an attribute named *Subscriber ID* is defined below in both the subscription and filter groups. The contents of this attribute are a handle provided by the front-end for its own use in implementing this local "routing" of indications to the management applications it supports. This handle is opaque to the DMI Service Provider at which the subscription and filter entries are established; the DMI Service Provider simply returns this handle as part of the indication information when it delivers the indication. **NOTE**: the implementation aspects of this opaque handle are purely a function of the implementation of the front-end e.g. persistence of the meaning of the handle over re-boots, management application crashes, etc.

The Indication Subscription group is defined next.

```
Name = "SP Indication Subscription"
Class = "DMTF|SP Indication Subscription|001"
Description = "This group defines the subscription information for a managing node"
              "interested in indications from this system.  The DMI Service Provider"
    "will maintain this as a table, with each row representing an individual"
    "managing node."
Key = 1,2,3,4
```

3.3.1.1 SUBSCRIBER RPC TYPE

```
Name = "Subscriber RPC Type"
ID = 1
Description = "This is an identifier of the type of RPC in use by the Subscriber.”
Access = Read-Write
Storage = Common
Type = String(64)

// NOTE:   the allowable RPC strings are defined as follows
//   “DCE RPC”
//   “ONC RPC”
//   “TI  RPC”
Value = unknown
```

3.3.1.2 SUBSCRIBER TRANSPORT TYPE

```
Name = "Subscriber Transport Type"
ID = 2
Description = "This is an identifier of the type of Transport in use by the Subscriber.”
Access = Read-Write
Storage = Common
Type = String(64)
Value = unknown
```

TRANSPORT NAME	DESCRIPTION
ncacn_nb_tcp	Connection-oriented NetBIOS over TCP
ncacn_nb_ipx	Connection-oriented NetBIOS over IPX
ncacn_nb_nb	Connection-oriented NetBEUI
ncacn_ip_tcp	Connection-oriented TCP/IP
ncacn_np	Connection-oriented named pipes
ncacn_spx	Connection-oriented SPX
ncacn_dnet_nsp	Connection-oriented DECnet
ncacn_at_dsp	Connection-oriented AppleTalk DSP
ncadg_ip_udp	Datagram (connectionless) UDP/IP
ncadg_ipx	Datagram (connectionless) IPX
ncalrpc	Local procedure call

3.3.1.3 SUBSCRIBER ADDRESSING

The format of the Subscriber Addressing field varies according to RPC type, Transport type, and the implementation of the Service Provider. For example, for DCE RPC and transport type ncacn_ip_tcp, the subscriber addressing information might take the form:

ipaddress [port number]

where ipaddress is in dotted decimal form, and port number is the TCP/IP port assigned to the management process during its initialization.

Because the format of this field is dependent on the Service Provider implementation, it is not possible to list the formats for each combination of RPC and Transport type here. In order to remove the burden of determining the correct contents and format of this field from the management application, SP vendors provide a support function called DmiGetSubscriptionAddress(). This function may be called by a management application to

obtain the subscriber addressing information for a given combination of RPC and Transport types. It takes the form:

```
DmiErrorStatus_t DMI_API
DmiGetSubscriptionAddress (
    [in] DmiString_t*   rpcType,
    [in] DmiString_t*   transportType,
    [out] DmiString_t*  address );


Name = "Subscriber Addressing"
ID = 3
Description = "Addressing information of the managing node that has subscribed”
              “to receive indications from this managed node."
Access = Read-Write
Storage = Common
Type = String(1024)
Value = unknown
```

3.3.1.4 SUBSCRIBER ID

```
Name = "Subscriber ID"
ID = 4
Description = "An ID or handle passed by the managing node to the SP.  It is opaque”
              “to the DMI Service Provider, but is used in all indications to the”
     “managing node as a correlator, or multiplexing handle.  It is intended”
     “only for use by the managing node.”
Access = Read-Only
Storage = Specific
Type = Integer
Value = unknown
```

3.3.1.5 SUBSCRIPTION EXPIRATION WARNING DATE STAMP

```
Name = "Subscription Expiration Warning Date Stamp"
ID = 5
Description = "On this date and time, the DMI Service Provider will send an”
              “indication to the subscriber, notifying it that the subscription”
              “is about to lapse.”

//  NOTE  If the transmission was UNSUCCESSFUL the DMI Service Provider
//  should reset this value using the following formula:
//    (((Exp TimeStamp)-(Warn Timestamp)) / 2) + (Warn Timestamp)
//  This behavior should continue until the indication is successfully
//  transmitted, or until either the Expiration date is reached, or the
//  Indication Failure Threshold is reached.

Access = Read-Only
Storage = Specific
Type = Date
Value = unknown
```

3.3.1.6 SUBSCRIPTION EXPIRATION DATESTAMP

```
Name = “Subscription Expiration DateStamp”
ID = 6
Description = "On this date, after having issued the appropriate number of”
              “warning indications as described by the Subscription Expiration”
              “Warning Timestamp, this subscription will lapse.”
              “NOTE: that then, this entry is to be removed by the DMI Service”
              “Provider, along with any filter table entries associated with it.”
Access = Read-Only
Storage = Specific
Type = DATE
Value = unknown
```

3.3.1.7 INDICATION FAILURE THRESHOLD

```
Name = "Indication Failure Threshold"
ID = 7
Description = "This is a number that corresponds to the number of indication"
              "transmission failures to allow, before the indication subscription"
              "is considered to be invalid, and removed."

Access = Read-Only
Storage = specific
Type = Integer
Value = unknown
```

3.3.2 SP Filter Information

This tabular group will be instantiated and maintained by the DMI Service Provider. It is a list of filters to applied to all outbound indications that are classified as events.

Filter operation

The operation of the filter is such that the event will pass, ie. will be forwarded to the managing node, if a filter is present that matches the event's ComponentID, Class string, and the event's severity is one of the severity levels specified in the *Event Severity* attribute.

Specifying a Component ID of 0xFFFFFFFF in the filter will match any component ID in the event. Specific component ID's may be used to match events generated by the corresponding component. Recall also that a component ID of zero implies that the event is being reported by an Event Reporter on the originating node that is not registered as a component with its DMI Service Provider.

Class strings may be matched by providing partial class strings in the filter in a manner similar to the class string parameter to the ListComponentsByClass command in the MI. For example, the partial class string "DMTF||001" will match all DMTF defined version 1 standard groups. Similarly, "||" will match all group definitions of all versions, whether defined by the DMTF or another other industry body or vendor. Likewise "|Processor|" will match all *Processor* groups of all versions whether defined by the DMTF or any other entity.

Event severity is matched by providing, in effect, a bit mask. It will be noted that the enumeration specifying event severity has been deliberately defined with selectors that are powers of 2. Thus to match multiple event severities a bit mask must be created by OR'ing the respective selectors. This bit mask is then stored in the *Event Severity* attribute in the filter entry and must be specially interpreted by management applications and service providers, namely:

- Management applications must not use the contents of the *Event Severity* attribute as simply a single enumeration selector but rather recognize that it is a bit mask and break it down into the corresponding event severities before printing it or otherwise manipulating it.
- DMI Service Providers must interpret the contents of *Event Severity* attribute as a bit mask rather than as a single enumeration selector when determining whether or not the event is to be propagated onto the communication network.

The SP Filter Information group is defined next:

```
Name = "SP Filter Information"
Class = "DMTF|SPFilterInformation|001"
Description = "This group defines a row in a table of event filters.  One filter"
              "is created for each combination of ComponentID, Class, and severity"
     "that the managing node is interested in."
Key = 1,2,3,4,5,6
```

3.3.2.1 SUBSCRIBER RPC TYPE

```
Name = "Subscriber RPC Type"
ID = 1
Description = "This is an identifier of the type of RPC in use by the Subscriber."
Access = Read-Write
Storage = Common
Type = String(64)

// NOTE:   the allowable RPC strings are defined as follows
//   "DCE RPC"
//   "ONC RPC"
```

```
//   "TI  RPC"
Value = unknown
```

3.3.2.2 SUBSCRIBER TRANSPORT TYPE

```
Name = "Subscriber Transport Type"
ID = 2
Description = "This is an identifier of the type of Transport in use by the Subscriber."
Access = Read-Write
Storage = Common
Type = String(64)
Value = unknown
```

TRANSPORT NAME	DESCRIPTION
ncacn_nb_tcp	Connection-oriented NetBIOS over TCP
ncacn_nb_ipx	Connection-oriented NetBIOS over IPX
ncacn_nb_nb	Connection-oriented NetBEUI
ncacn_ip_tcp	Connection-oriented TCP/IP
ncacn_np	Connection-oriented named pipes
ncacn_spx	Connection-oriented SPX
ncacn_dnet_nsp	Connection-oriented DECnet
ncacn_at_dsp	Connection-oriented AppleTalk DSP
ncadg_ip_udp	Datagram (connectionless) UDP/IP
ncadg_ipx	Datagram (connectionless) IPX
ncalrpc	Local procedure call

3.3.2.3 SUBSCRIBER ADDRESSING

```
Name = "Subscriber Addressing"
ID = 3
Description = "Addressing information of the managing node that has subscribed"
              "to receive indications from this managed node."
Access = Read-Write
Storage = Common
Type = String(1024)
Value = unknown
```

3.3.2.4 SUBSCRIBER ID

```
Name = "Subscriber ID"
ID = 4
Description = "An ID or handle passed by the managing node to the SP.  It is"
     "opaque to the DMI Service Provider, but is used in all"
     "indications to the managing node as a correlator, or"
              "multiplexing handle.  It is intended only for use by the"
              "managing node."
Access = Read-Only
Storage = Specific
Type = Integer
Value = unknown
```

3.3.2.5 COMPONENT ID

```
Name = "Component ID"
ID = 5
Description = "The component ID, as assigned by the DMI Service Provider, of the”
              “component from which the managing node wishes to receive events.”
Access = Read-Write
Storage = Specific
Type = Integer
Value = unknown
```

3.3.2.6 GROUP CLASS STRING

```
name = "Group Class String"
ID = 6
Description = "The Class string corresponding to the groups within the above”
              “mentioned component, from which the managing node wishes to”
              “receive events.”
Access = Read-Write
Storage = Specific
Type = String(64)
Value = unknown

//  Note: that a value of NULL STRING should be used if the entity generating
//  this event is an application.
```

3.3.2.7 EVENT SEVERITY

This particular attribute within a row of the SP Filter Information Entry group needs to be treated specially by Management Applications (i.e. subscribers for event notifications) and by DMI Service Providers. The Event Severity enumeration is purposely defined as a bit-mask so that multiple event severities may be selected for a filter entry. This means that when a management application reads a row of this group it must be aware that the contents of this attribute might be a set of enumeration selectors that have been OR'ed together. In other words, the contents of this attribute in the entry should not automatically be treated as a single enumeration selector as would happen in the case of normal enumerations. DMI Service Providers must also interpret the contents of this attribute as potentially a set of OR'ed enumeration selectors that specify several event severities for filtering.

```
Name = "Event Severity"
ID = 7
Description = "The event severity level, at which an event originating ”
     “in a group described by the previous class and componentID, should be ”
     “forwarded to the managing node. Note that ”
     "The Severity enumeration is defined as a bit mask so that events at more "
     "than one level of Severity may be requested by OR'ing together the appropriate "
     "Severity selectors."
Type = Start Enum
              0x001 = “Monitor”
              0x002 = “Information”
              0x004 = “OK”
              0x008 = “Non-Critical”
              0x010 = “Critical”
              0x020 = “Non-Recoverable”
      End Enum
Access = Read-Write
Storage = Specific
Value = unknown
```

3.4 EVENT EXAMPLE

This section uses the previously described event model with standard groups to demonstrate the construction of an Event Generation group.

Assume that a spreadsheet product has two executable modules: *file.exe* and *calc.exe*. File.exe opens and closes worksheets and calc.exe performs calculations on them. Each of the modules can fault in various ways: (1) File.exe can encounter a read error or a write error. (2) Calc.exe can encounter an overflow error or an out of range error. In addition, calc.exe can encounter a write error during an automatic save.

3.4.1 Software Signature Template[6]

```
Start Group
Name = "Software Signature"
Class = "DMTF|Software Signature|001"
Key = 1

    Start Attribute
    Name = "File Name"
    ID = 1
    Storage = Common
    Access = Read-Only
    Type = String(256)
    End Attribute

    Start Attribute
    Name = "File Size"
    ID = 2
    Storage = Specific
    Access = Read-Only
    Type = Integer
    End Attribute

    Start Attribute
    Name = "File Date and Time"
    ID = 3
    Storage = Specific
    Access = Read-Only
    Type = Date
    End Attribute

    Start Attribute
    Name = "File Checksum"
    ID = 4
    Storage = Specific
    Access = Read-Only
    Type = Integer
    End Attribute

    Start Attribute
    Name = "File CRC 1"
    ID = 5
    Access = Read-Only
    Type = Integer
    End Attribute

    Start Attribute
    Name = "File CRC 2"
    ID = 6
    Storage = Specific
    Access = Read-Only
    Type = Integer
    End Attribute
End Group
```

[6] The groups in this section are reproduced without the descriptions for the sake of brevity. For the same reason, the ComponentID group and Software Component Information group are not reproduced here.

3.4.2 Software Signature Table[7]

```
Start Table
Name = "Software Signature"
Class = "DMTF|Software Signature|001"
ID = 3[8]

{"file.exe", 100, "19950101000000.000000-000", 200, 300, 400}
{"calc.exe", 100, "19950101000000.000000-000", 200, 300, 400}
End Table
```

3.4.3 Event Generation Group

```
Start Enum
Name = "BOOL"
     0 = "False"
     1 = "True"
End Enum
Start Group
Name = "Event Generation"
Class = "EventGeneration|DMTF^^Software Signature Example|002"
ID = 4
Key = 5

     Start Attribute
     Name = "Event Type"
     ID = 1
     Type = Start Enum
            1 = "Read Error"
            2 = "Write Error"
            3 = "Out of Range"
            4 = "Overflow"
            End Enum
     Access = Read-Only
     Storage = Specific
     Value = unknown
     End Attribute

     Start Attribute
     Name = "Event Severity"
     ID = 2
     Type = Start Enum
            0x001 = "Monitor"
            0x002 = "Information"
            0x004 = "OK"
            0x008 = "Non-Critical"
            0x010 = "Critical"
            0x020 = "Non-Recoverable"
            End Enum
     Access = Read-Only
     Storage = Specific
     Value = unknown
     End Attribute

     Start Attribute
     Name = "Event Is State-Based"
     ID = 3
     Type = "BOOL"
     Access = Read-Only
     Storage = Specific
     Value = unknown
     End Attribute

     Start Attribute
     Name = "Event State Key"
     ID = 4
     Type = Integer
     Access = Read-Only
     Storage = Specific
     Value = unknown
```

[7] The values of the numeric data in this table are contrived.
[8] ID 1 is the ComponentID group. ID 2 is the Software Component Information group.

```
        End Attribute

        Start Attribute
        Name = "Associated Group"
        ID = 5
        Type = String
        Access = Read-Only
        Storage = Common
        Value = "DMTF|Software Signature|001"
        End Attribute

        Start Attribute
        Name = "Event System"
        ID = 6
        Type = Start Enum
               1 = "I/O"
               2 = "Calculation"
               End Enum
        Access = Read-Only
        Storage = Specific
        Value = unknown
        End Attribute

        Start Attribute
        Name = "Event Subsystem"
        ID = 7
        Type = Start Enum
               0 = "None"
               End Enum
        Access = Read-Only
        Storage = Specific
        Value = unknown
        End Attribute

        Start Attribute
        Name = "Instance Is Data Present"
        ID = 8
        Type = "BOOL"
        Access = Read-Only
        Storage = Specific
        Value = "False"
        End Attribute
    End Group
```

3.4.4 MIF Template

```
    ////////////////////////////////////////
    // DMTF Standard Event Group Definition //
    ////////////////////////////////////////

    ////////////////////////
    // Common Definitions //
    ////////////////////////

    Start Enum
    Name = "BOOL"
         0 = "False"
         1 = "True"
    End Enum



////////////////////////////////////////////////////////////
// Group Definition                                       //
// (Replace bracketed identifiers with actual definition.) //
////////////////////////////////////////////////////////////

    Start Group
    Name = "Event Generation"
    Class = "EventGeneration|<Specific name>|002"
    ID = <ID>
    Key = 5
```

```
////////////////////////
// Required Attributes //
////////////////////////

            Start Attribute
            Name = "Event Type"
            ID = 1
            Description = "The type of event that has occurred."
            Type = <Enum>
            Access = Read-Only
            Storage = Specific
            Value = unknown  // Value definition required by Installer.  Ignore.
            End Attribute

            Start Attribute
            Name = "Event Severity"
            ID = 2
            Description = "The severity of this event."
            Type = Start Enum
                   0x001 = "Monitor"
                   0x002 = "Information"
                   0x004 = "OK"
                   0x008 = "Non-Critical"
                   0x010 = "Critical"
                   0x020 = "Non-Recoverable"
                   End Enum
            Access = Read-Only
            Storage = Specific
            Value = unknown  // Value definition required by Installer.  Ignore.
            End Attribute

            Start Attribute
            Name = "Event Is State-Based"
            ID = 3
            Description = "The value of this attribute determines"
            "whether the Event being reported is a"
                          "state-based Event or not. If the value of"
            "this attribute is TRUE then the Event is "
                          "state-based. Otherwise the Event is not "
                          "state-based."
            Type = "BOOL"
            Access = Read-Only
            Storage = Specific
            Value = unknown  // Value definition required by Installer.  Ignore.
            End Attribute

            Start Attribute
            Name = "Event State Key"
            ID = 4
            Description = "A unique, single integer key into the”
            "Event State group if this is a state-based"
            "Event. If this is not a state-based Event then”
            “this attribute's value is not defined."
            Type = Integer
            Access = Read-Only
            Storage = Common
            Value = unknown  // Value definition required by Installer.  Ignore.
            End Attribute
```

```
  Start Attribute
  Name = "Associated Group"
  ID = 5
  Description = "The class name of the group that is associated”
                “with the events defined in this Event Generation”
                 “group."
  Type = String
  Access = Read-Only
  Storage = Common
  Value = "<Class name>"
  End Attribute

  Start Attribute
  Name = "Event System"
  ID = 6
  Description = "The major functional aspect of the product causing”
                “the fault."
  Type = <Enum>
  Access = Read-Only
  Storage = Specific
  Value = unknown  // Value definition required by Installer.  Ignore.
  End Attribute

  Start Attribute
  Name = "Event Subsystem"
  ID = 7
  Description = "The minor functional aspect of the"
                "product causing the fault."
  Type = <Enumeration>
  Access = Read-Only
  Storage = Specific
  Value = unknown  // Value definition required by Installer.  Ignore.
  End Attribute

//////////////////////////
// Optional Attributes //
//////////////////////////

  Start Attribute
  Name = "Event Solution"
  ID = 8
  Description = "A solution to the problem that caused the event."
  Type = <Enum>
  Access = Read-Only
  Storage = Specific
  Value = unknown  // Value definition required by Installer.  Ignore.
  End Attribute

  Start Attribute
  Name = "Instance Data Present"
  ID = 9
  Description = "Indicates whether the second event"
                "data structure contains instance-specific data."
  Type = "BOOL"
  Access = Read-Only
  Storage = Specific
  Value = unknown  // Value definition required by Installer.  Ignore.
  End Attribute
```

```
                Start Attribute
                Name = "Vendor Specific Message"
                ID = 10
                Description = "Auxiliary information related to the event."
                Type = String(<Size>)
                Access = Read-Only
                Storage = Specific
                Value = unknown  // Value definition required by Installer.  Ignore
                End Attribute

                Start Attribute
                Name = "Vendor Specific Data"
                ID = 11
                Description = "Auxiliary information related to the event."
                Type = OctetString(<Size>)
                Access = Read-Only
                Storage = Specific
                Value = unknown  // Value definition required by Installer.  Ignore
                End Attribute

End Group
```

4. INTERFACE OVERVIEW

In the DMI framework there are four broad classes of APIs as depicted abstractly in Figure 4-1. They are, respectively,

Management Application Provider Functions. These are functions implemented by the Management Application Provider that may be invoked by the DMI Service Provider. An example of this is the function entry point at which the DMI Service Provider delivers Indications to the Management Application. The Management Application Provider Functions are specified in Section 7.

DMI Service Provider Functions for Management Applications. These are functions implemented by the DMI Service Provider that may be invoked by Management Applications. All of the functions in the DMI Service Provider Functions for Management Applications are specified as part of the Management Interface (MI) in subsequent Section 6.

DMI Service Provider Functions for Components. These are functions implemented by the DMI Service Provider that may be invoked by Component Providers. Registrations functions, or Indication origination functions fall into this abstract class. The DMI Service Provider Functions for Components are specified as part of the Component Interface (CI) in Section 8. These functions are OS-specific. Some OSes may not implement the CI but provide the equivalent functionality using other, native mechanisms.

Component Provider Functions. These are functions implemented by Component Providers that may be invoked by DMI Service Providers. Examples of these functions are CiGetAttribute and CiSetAttribute. The Component Provider functions are specified as part of the Component Interface (CI) in Section 8. These functions are OS-specific. Some OSes may not implement the CI but provide the equivalent functionality using other, native mechanisms.

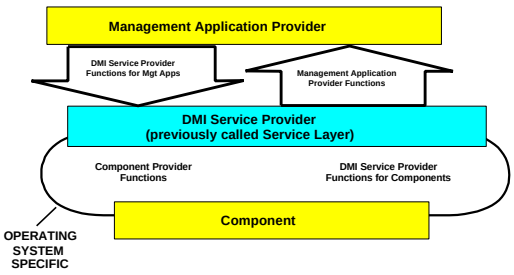


Figure 4-1. Abstract classes of APIs in the DMI Framework.

In this document the DMI Service Provider Functions for Management Applications are defined in Section 6 "Management Interface". The Management Application Provider functions are defined in Section 7 "Management Application Provider API". The remaining two abstract classes of functions described above are defined in Section 8 "Component Interface".

4.1 PROGRAMMING CONSIDERATIONS

Working in an RPC environment has some unusual characteristics that merit special attention. The following section introduces some of these issues. However, a complete discussion of all RPC issues is outside the scope of this document. Appendix D contains a list of related documents for further reference.

4.1.1 Binding To A Managed Machine

One of the first questions to answer when developing a management application is that of connecting, or *binding*, to the managed machine. The DMI 2.0 interface relies on standard RPC mechanisms to accomplish this binding.

To connect to a machine, a management application must supply

- the machine's name or address,
- the protocol sequence (e.g,, TCP/IP),
- the Service Provider's process address (endpoint) on the managed machine,
- and the user's identity

A management application will typically specify the machine name and protocol sequence, and will most likely use a dynamically determined endpoint. This addressing data is used to construct a *binding handle;* binding handles are RPC-defined data structures that are used to manage the connection between RPC clients and servers.

Management applications that only talk to *one machine at a time* can construct an *implicit*, or global, binding handle. When used in this manner, the application is effectively saying that all remote procedure calls are directed toward a specific machine. When the application is done talking to that machine, it will free the binding. At this point, the application can construct a new binding handle for some other machine.

Management applications that *simultaneously* manage multiple machines will need to construct and maintain multiple binding handles: one per connection. In this usage model, the management application must explicitly supply a binding handle with each procedure call. This allows an application to direct procedure calls to different machines, while eliminating the need to create and free binding handles between procedure calls.

The Management Interface APIs specified in Sections 6 and 7 ***do not*** include binding handles in the procedures' formal parameter lists. Instead, these API specifications concentrate on the DMI 2.0 interfaces themselves.

Some RPC implementations can retrieve the management application's user identity implicitly and provide it to the managed machine Service Provider.

NOTE: The absence of a binding handle in a procedure's formal parameter list does not preclude the use of explicit binding handles in a management application. The DCE RPC programming environment, for example, provides a mechanism whereby management applications can tailor the interface for *implicit* or *explicit* binding, without changing the IDL description itself. This customization occurs when the developer creates the RPC procedure stubs with the RPC IDL compiler. Appendix B describes the DCE RPC development process and includes the DCE IDL description for the interfaces described in this document.

4.1.2 The use of pointers

In general, the formal parameter list for any procedure will be composed of three parameter types: in, out, and in/out. The "in" parameters are used to pass information to the procedure; the "out" parameters (including the procedure's return value) are used to return results from the procedure, and the "in/out" parameters are used to both pass information and to receive results.

For simple data types, we can pass the data *by value*. This is the case, for example, when passing the component ID to a procedure. To receive a simple data type in return, the caller passes the address of a variable to hold the result.

When a procedure call returns from a remote system, the RPC stub copies the data value into the address specified by the caller.

Things become a little more complicated when passing data structures *by reference*. The DMI procedural interface contains procedures that accept and return arrays of data structures. These structures are passed by reference, with some memory allocated by the management application, and some allocated by the DMI Service Provider. Given all this memory allocation, we need some clear rules about who performs the allocation, and who owns the allocated data. For each parameter class, the responsibility for allocating and freeing reference parameters is as follows:

TYPE	ALLOCATI D BY	OWNED BY
In	Caller	Caller
Out	Callee	Caller
In/Out	Caller on input; callee reallocates on output	Caller

In the latter two cases there is one piece of code (e.g., the RPC stub) that allocates the memory and a different piece of code (e.g., the management application) that frees it. For this to be successful, the two pieces of code must have knowledge of which memory allocator is being used. In RPC programming environments, the client application and the RPC stubs use a common memory allocator, usually specified by the RPC runtime system.

Further, the treatment of out and in-out parameters in failure conditions requires special attention. If a function returns a status code which is a failure code, then in general the caller has no way to clean up the *out* or *in-out* parameters returned to him. This leads to a few additional rules:

out parameters

For error returns, out parameters must be *always* reliably set to a value which will be cleaned up without any action on the caller's part.

Further, it is the case that all out pointer parameters (usually passed in a pointer-to-pointer parameter, but which can also be passed as a member of a caller-allocate, callee-fill structure) *must* explicitly be set to NULL.

As a DMI management application writer, then, you should assume that a failed procedure call requires no additional memory cleanup; the DMI Service Provider should NOT allocate any memory in the failure case.

in-out parameters

For error returns, all in-out parameters must either be left alone by the callee (and thus remaining at the value to which it was initialized by the caller) or be explicitly set as in the out parameter error return case.

4.1.3 Calling Conventions

In order to support portability, and for clarity in this document, all of the DMI functions are defined to have a calling convention of DMI_API.

For example:

```
DmiErrorStatus_t DMI_API DmiAddRow(DmiHandle_t Handle, DmiRowData_t *RowData);
```

This allows a calling convention that is native to a host operating system to be used when building implementations for that operating system. The following is a list of calling conventions to be used by each of the Operating Systems discussed in this document:

OS	IMPLEMENTATIONS
macos	
Os2	#define DMI_API APIENTRY
unix	
win16	#define DMI_API WINAPI
win32	#define DMI_API WINAPI
win9x	#define DMI_API WINAPI
winnt	#define DMI_API WINAPI

4.1.4 Re-entrancy

Most, if not all, 32-bit operating system environments today provide multi-threaded operation. In addition, in a networked environment, there may be several simultaneous sources of function calls to any particular function entry point. In consequence, all entry points in the procedural interface portion of this specification must be implemented to be re-entrant, with the exception of the Component Provider functions. This exception is provided to subsume current implementations of component instrumentation code with a minimum of re-design.

4.2 NATIONAL LANGUAGE SUPPORT

4.2.1 Requirement

The DMI has always supported NLS functionality, but with this version it is no longer an optional element. Any implementation that claims to be conformant to this specification MUST support all of the NLS functions defined in this specification. One important note for component vendors, with this version of the specification the LANGUAGE statement, as defined in Section 2.2 (MIF Grammar) of this document, is no longer optional.

4.2.2 Overview

DMI handles NLS functionality through several functions defined in this document. This section presents a brief overview of all of those functions. There are two primary mechanisms that are enabled in the DMI architecture that allow for NLS to work. The first is the installability of additional MIF files, known as language mapping files. These files are MIF files that differ in two ways - the language string at the top (which is now mandatory in all MIF files) defines the language and encoding style used for this file, and secondly that the translatable text is in that language. The second mechanism defined in this spec to enable NLS is the use of two different character encoding styles. This document allow the use of either ISO 8859-1 (Latin Alphabet I) for those languages that can be represented using this single byte character set, or UNICODE. UNICODE is a two byte character set that represents an attempt to combine the multitude of character sets, and encoding styles into a single element. It should be noted that the first 255 code points of the UNICODE code page correspond exactly to ISO 8859-1, so coexistence is greatly simplified.

NOTE: the above description refers to OS environments that implement the CI interface described in Section 8. However, the functionality and database schema implied by the CI are OS-specific. Some OSes may not implement the CI functions and the MIF schema but provide equivalent functionality using other, native mechanisms and native schema's. In this case the language mapping files are another form of schema description files in that environment.

4.2.3 Translatable Text

A discussion of what is translatable within a MIF file is probably best dealt with by stating what is NOT translatable within a MIF file. The following is a list of the MIF elements that are NOT translatable:

1) Keywords
2) Language strings
3) Class strings
4) String values that are keys

4.2.4 Installation

As stated above, NLS support is initiated by the installation of multiple MIF files for a given component. This is accomplished by use of the **DmiAddComponent()** and **DmiAddLanguage()** functions. The primary difference between these functions is that one - **DmiAddComponent()** returns a component ID, and the other **DmiAddLanguage()** takes a component ID as one of its input parameters.

It should be noted, that DmiAddComponent() can be used to install both the Default MIF and language mapping MIFs all at the same time. This is done through the use of the DmiFileDataList_t data structure. The first, or only MIF file passed to DmiAddComponet() will become the default language for that component, and any additional MIF files (and all files passed to DmiAddLanguage()) will be used as requestable languages. Additional languages can be installed for a given component at any time, but it should be noted that since Groups can be added to, or removed from, a component at any time, the newly installed language mapping should make a reasonable attempt to match the installed component.

NOTE: the above description refers to OS environments that implement the CI interface described in Section 8. However, the functionality and database schema implied by the CI are OS-specific. Some OSes may not implement the CI functions and MIF schema but provide the equivalent functionality using other, native mechanisms and native schemas. Also see Section 6.4.

4.2.5 Operation

In operation, the DMI allows a user to discover and select the language to use on all subsequent requests in the following manner. A user of the MI interface can issue the **DmiListLanguages()** to retrieve a list of the languages that are currently available for a given component. The DMI Service Provider will return queries to all commands using the default (first) language installed for a component, unless or until the application uses the **DmiSetConfig()** function to change the response language. An application can issue this call at any time, and as often as needed, but it should be noted that for the periods between invocations of this function, all DMI functions will use the currently set language to build responses. If a component does not have the requested language installed to support a given request, then the DMI Service Provider will use the default (first) language for the response, and an error code of **DMIERR_DEFAULT_LANGUAGE_RETURNED** will be returned to the caller.

5. KEY DATA STRUCTURES

5.1 DMI DATA TYPES

The DMI data types presented in this specification adhere to the naming convention for DCE RPC data types. DCE data types have the following size representations:

IDL Datatype	Size
char	8 bits
boolean	8 bits
long	32 bits
hyper	64 bits
unsigned long	32 bits
unsigned hyper	64 bits

```
typedef unsigned long      DmiCounter_t;
typedef unsigned hyper     DmiCounter64_t;
typedef unsigned long      DmiErrorStatus_t;
typedef unsigned long      DmiGauge_t;
typedef unsigned long      DmiHandle_t;
typedef unsigned long      DmiId_t;
typedef long               DmiInteger_t;
typedef hyper              DmiInteger64_t;
typedef unsigned long      DmiUnsigned_t;
typedef boolean            DmiBoolean_t;
```

5.2 ENUMERATED TYPES

5.2.1 DmiAccessMode

This enumerated type defines the access modes for an attribute.

FIELD NAME	DESCRIPTION
MIF_UNKNOWN	Unknown access mode
MIF_READ_ONLY	Read access only
MIF_READ_WRITE	Readable and writable
MIF_WRITE_ONLY	Write access only
MIF_UNSUPPORTED	Attribute is not supported

```
typedef enum {
    MIF_UNKNOWN,
    MIF_READ_ONLY,
    MIF_READ_WRITE,
    MIF_WRITE_ONLY,
    MIF_UNSUPPORTED
} DmiAccessMode_t;
```

5.2.2 DmiDataType

This enumerated type defines the data types referenced by DmiDataUnion.

FIELD NAME	DESCRIPTION
MIF_DATATYPE_0	RESERVED
MIF_COUNTER	32-bit unsigned integer that never decreases
MIF_COUNTER64	64-bit unsigned integer that never decreases
MIF_GAUGE	32-bit unsigned integer that may increase or decrease
MIF_DATATYPE_4	RESERVED
MIF_INTEGER	32-bit signed integer
MIF_INTEGER64	64-bit signed integer
MIF_OCTETSTRING	String of n octets, not necessarily displayable
MIF_DISPLAYSTRING	Displayable string of n octets
MIF_DATATYPE_9	RESERVED
MIF_DATATYPE_10	RESERVED
MIF_DATE	28-octet displayable string (`yyyymmddhhmmss.uuuuuu+ooo`)

```
typedef enum {
    MIF_DATATYPE_0,
    MIF_COUNTER,
    MIF_COUNTER64,
    MIF_GAUGE,
```

```
    MIF_DATATYPE_4,
    MIF_INTEGER,
    MIF_INTEGER64,
    MIF_OCTETSTRING,
    MIF_DISPLAYSTRING,
    MIF_DATATYPE_9,
    MIF_DATATYPE_10,
    MIF_DATE
} DmiDataType_t;
```

5.2.3 DmiFileType

This data structure defines the DMI mapping file types.

FIELD NAME	DESCRIPTION
DMI_FILETYPE_0	RESERVED
DMI_FILETYPE_1	RESERVED
DMI_MIF_FILE_NAME	File data is the name of a DMI MIF file
DMI_MIF_FILE_DATA	File data is the contents of DMI MIF file
SNMP_MAPPING_FILE_NAME	File data is the name of an SNMP mapping file
SNMP_MAPPING_FILE_DATA	File data is the contents of an SNMP mapping file
DMI_GROUP_FILE_NAME	File data is the name of a DMI GROUP file
DMI_GROUP_FILE_DATA	File data is the contents of a DMI GROUP file
VENDOR_FORMAT_FILE_NAME	File data is the name of a Vendor-format data file
VENDOR_FORMAT_FILE_DATA	File data is the contents of a Vendor-format data file

```
typedef enum {
    DMI_FILETYPE_0,
    DMI_FILETYPE_1,
    DMI_MIF_FILE_NAME,
    DMI_MIF_FILE_DATA,
    SNMP_MAPPING_FILE_NAME,
    SNMP_MAPPING_FILE_DATA,
    DMI_GROUP_FILE_NAME,
    DMI_GROUP_FILE_DATA,
    VENDOR_FORMAT_FILE_NAME,
    VENDOR_FORMAT_FILE_DATA
} DmiFileType_t;
```

5.2.4 DmiRequestMode

This data structure defines sequential access modes.

FIELD NAME	DESCRIPTION
DMI_UNIQUE	Access the specified item (or table row)
DMI_FIRST	Access the first item
DMI_NEXT	Access the next item

```
typedef enum {
    DMI_UNIQUE,
    DMI_FIRST,
```

```
    DMI_NEXT
} DmiRequestMode_t;
```

5.2.5 DmiSetMode

This data structure describes set operations.

FIELI NAME	DESCI IPTION
DMI_SET	Set data values
DMI_RESERVE	Reserve resources for a set operation
DMI_RELEASE	Release previously reserved resources

```
typedef enum {
    DMI_SET,
    DMI_RESERVE,
    DMI_RELEASE
} DmiSetMode_t;
```

5.2.6 DmiStorageType

This data structure defines the storage type for an attribute.

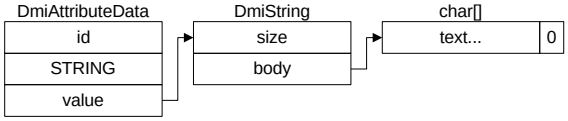
FIELI NAME	DESCI IPTION
MIF_COMMON	Value is from a small set of possibilities
MIF_SPECIFIC	Value is from a large set of possibilities

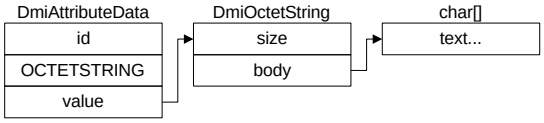
```
typedef enum {
    MIF_COMMON,
    MIF_SPECIFIC
} DmiStorageType_t;
```

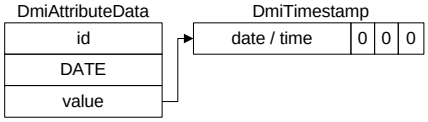
5.3 DATA STRUCTURES

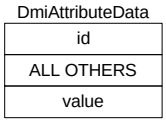
5.3.1 DmiAttributeData

This data structure describes an attribute id, type, and value.









FIELD NAME	DESCRIPTION
id	Attribute ID
data	Attribute type and value

```
typedef struct DmiAttributeData {
    DmiId_t                 id;
    DmiDataUnion_t          data;
} DmiAttributeData_t;
```

5.3.2 DmiAttributeIds

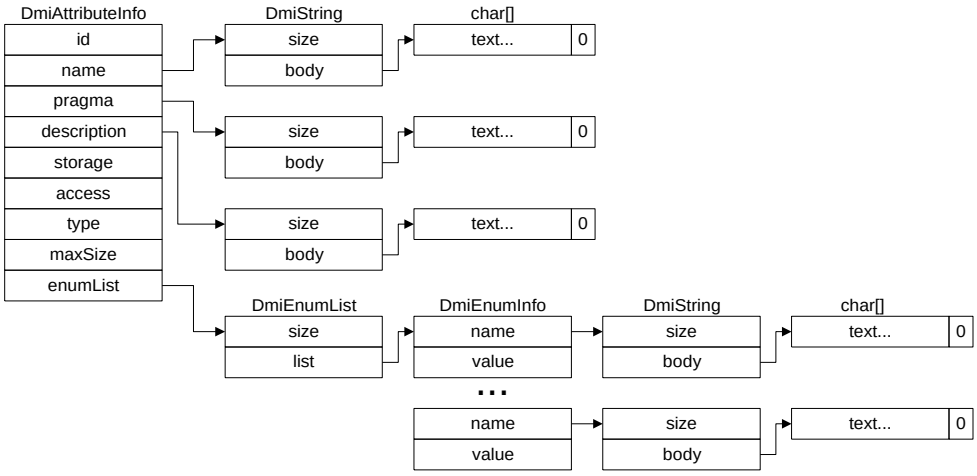
This data structure describes a conformant array of DmiAttributeIds.

FIELD NAME	DESCRIPTION
size	Array elements
list	Array data

```
typedef struct DmiAttributeIds {
    DmiUnsigned_t          size;
    DmiId_t*               list;
} DmiAttributeIds_t;
```

5.3.3 DmiAttributeInfo

This data structure holds information about an attribute.



FIELD NAME	DESCRIPTION
id	Attribute ID
name	Attribute name string
pragma	Attribute pragma string [optional]
description	Attribute description string [optional]
storage	Common or specific storage
access	read-only, read-write, etc.
type	Counter, integer, etc.
maxSize	Maximum length of the attribute
enumList	EnumList for enumerated types [optional]

```
typedef struct DmiAttributeInfo {
    DmiId_t         id;
    DmiString_t*            name;
    DmiString_t*            pragma;
    DmiString_t*            description;
    DmiStorageType_t        storage;
    DmiAccessMode_t                access;
    DmiDataType_t           type;
    DmiUnsigned_t           maxSize;
    struct DmiEnumList*     enumList;
} DmiAttributeInfo_t;
```

5.3.4 DmiAttributeList

This data structure describes a conformant array of DmiAttributeInfo structs.

FIELD NAME	DESCRIPTION
size	Array elements
list	Array data

```
typedef struct DmiAttributeList {
    DmiUnsigned_t           size;
    DmiAttributeInfo_t*     list;
} DmiAttributeList_t;
```

5.3.5 DmiAttributeValues

This data structure describes a conformant array of DmiAttributeValues.

FIELD NAME	DESCRIPTION
size	Array elements
list	Array data

```
typedef struct DmiAttributeValues {
    DmiUnsigned_t           size;
    DmiAttributeData_t*     list;
} DmiAttributeValues_t;
```

5.3.6 DmiClassNameInfo

This data structure holds a group's id and class string.

FIELD N \ME	DESCRIP] ION
id	Group ID
className	Group class name string

```
typedef struct DmiClassNameInfo {
    DmiId_t                 id;
    DmiString_t*            className;
} DmiClassNameInfo_t;
```

5.3.7 DmiClassNameList

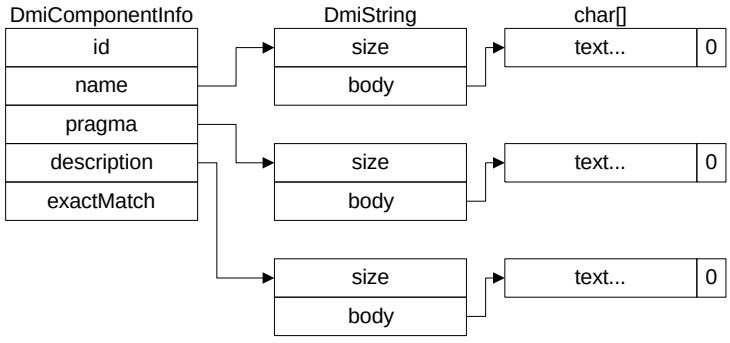
This data structure describes a conformant array of DmiClassNameInfo structs.

FIELD N \ME	DESCRIP] ION
size	Array elements
list	Array data

```
typedef struct DmiClassNameList {
    DmiUnsigned_t           size;
    DmiClassNameInfo_t*     list;
} DmiClassNameList_t;
```

5.3.8 DmiComponentInfo

This data structure holds information about a component.



FIEL) NAME	DESCRIPTION
id	Component ID
name	Component name string
pragma	Component pragma string [optional]
description	Component description string [optional]
exactMatch	TRUE = Exact match FALSE = Possible match

```
typedef struct DmiComponentInfo {
    DmiId_t                id;
    DmiString_t*           name;
    DmiString_t*           pragma;
    DmiString_t*           description;
    DmiBoolean_t           exactMatch;
} DmiComponentInfo_t;
```

5.3.9 DmiComponentList

This data structure describes a conformant array of DmiComponentInfo structs.

FI LD NAME	DESCRIPTI ON
size	Array elements
list	Array data

```
typedef struct DmiComponentList {
    DmiUnsigned_t          size;
    DmiComponentInfo_t*    list;
} DmiComponentList_t;
```

5.3.10 DmiDataUnion

This data structure is a discriminated union of DMI data types.

FI :LD NAME	DESCRIPTI ON
type	Discriminator for the union
value	Union of DMI attribute data types

```
typedef union
    switch (DmiDataType_t type) value {
    case MIF_COUNTER:              DmiCounter_t       counter;
    case MIF_COUNTER64:            DmiCounter64_t     counter64;
    case MIF_GAUGE:                DmiGauge_t         gauge;
    case MIF_INTEGER:              DmiInteger_t       integer;
    case MIF_INTEGER64:            DmiInteger64_t     integer64;
    case MIF_OCTETSTRING:          DmiString_t*       octetstring;
    case MIF_DISPLAYSTRING:        DmiString_t*       displaystring;
    case MIF_DATE:                 DmiTimestamp_t*    date;
} DmiDataUnion_t;
```

5.3.11 DmiEnumInfo

This data structure associates an integer value with descriptive text.

FIELD NAME	DESCRIPTION
name	Enumeration name
value	Enumeration value

```
typedef struct DmiEnumInfo {
    DmiString_t*           name;
    DmiInteger_t           value;
} DmiEnumInfo_t;
```

5.3.12 DmiEnumList

This data structure describes a conformant array of DmiEnumInfo structs.

FIELD NAME	DESCRIPTION
size	Array elements
list	Array data

```
typedef struct DmiEnumList {
    DmiUnsigned_t          size;
    DmiEnumInfo_t*         list;
} DmiEnumList_t;
```

5.3.13 DmiFileDataInfo

This data structure holds language file type and mapping data.

FIELD NAME	DESCRIPTION
fileType	Mif file, SNMP mapping file, etc.
file Data	The file info (name or contents)

```
typedef struct DmiFileDataInfo {
    DmiFileType_t          fileType;
    DmiOctetString_t*      fileData;
} DmiFileDataInfo_t;
```

5.3.14 DmiFileDataList

This data structure describes a conformant array of DmiFileDataInfo structs.

FIELD NAME	DESCRIPTION
size	Array elements
list	Array data

```
typedef struct DmiFileDataList {
    DmiUnsigned_t          size;
    DmiFileDataInfo_t*     list;
} DmiFileDataList_t;
```

5.3.15 DmiFileTypeList

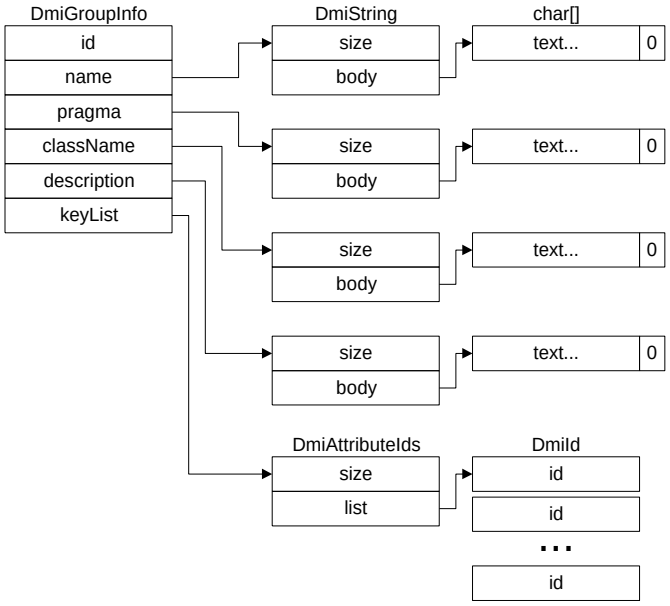
This data structure describes a conformant array of DmiFileTypes. It is used by the DmiGetVersion function to return a list of file types supported by the DmiAddComponent, DmiAddLanguage, and DmiAddGroup functions.

FIELD NAME	DESCRIPTION
size	Array elements
list	Array data

```
typedef struct DmiFileTypeList {
    DmiUnsigned_t          size;
    DmiFileType_t*         list;
} DmiFileTypeList_t;
```

5.3.16 DmiGroupInfo

This data structure holds information about a group.



FIEI D NAME	DESCRIP' 'ION
id	Group ID
name	Group name string
pragma	Group pragma string [optional]
className	Group class name string
description	Group description string [optional]
keyList	Attribute Ids for table row keys

```
typedef struct DmiGroupInfo {
    DmiId_t                 id;
    DmiString_t*            name;
    DmiString_t*            pragma;
    DmiString_t*            className;
    DmiString_t*            description;
    struct DmiAttributesIds*      KeyList;
} DmiGroupInfo_t;
```

5.3.17 DmiGroupList

This data structure describes a conformant array of DmiGroupInfo structs.

FIELD NAME	DESCRIPTION
size	Array elements
list	Array data

```
typedef struct DmiGroupList {
    DmiUnsigned_t          size;
    DmiGroupInfo_t* list;
} DmiGroupList_t;
```

5.3.18 DmiMultiRowData

This data structure describes a conformant array of DmiRowData structs.

FIELD NAME	DESCRIPTION
size	Array elements
list	Array data

```
typedef struct DmiMultiRowData {
    DmiUnsigned_t          size;
    DmiRowData_t*          list;
} DmiMultiRowData_t;
```

5.3.19 DmiMultiRowRequest

This data structure describes a conformant array of DmiRowRequest structs.

FIELD NAME	DESCRIPTION
size	Array elements
list	Array data

```
typedef struct DmiMultiRowRequest {
    DmiUnsigned_t          size;
    DmiRowRequest_t*       list;
} DmiMultiRowRequest_t;
```

5.3.20 DmiNodeAddress

This data structure describes addressing information for indication originators.

FIELD NAME	DESCRIPTION
address	Transport-dependent node address
rpc	Identifies the RPC (DCE, ONC, etc)
transport	Identifies the transport (TCP/IP, SPX, etc.)

```
typedef struct DmiNodeAddress {
    DmiString_t*           address;
    DmiString_t*           rpc;
    DmiString_t*           transport;
} DmiNodeAddress_t;
```

5.3.21 DmiOctetString

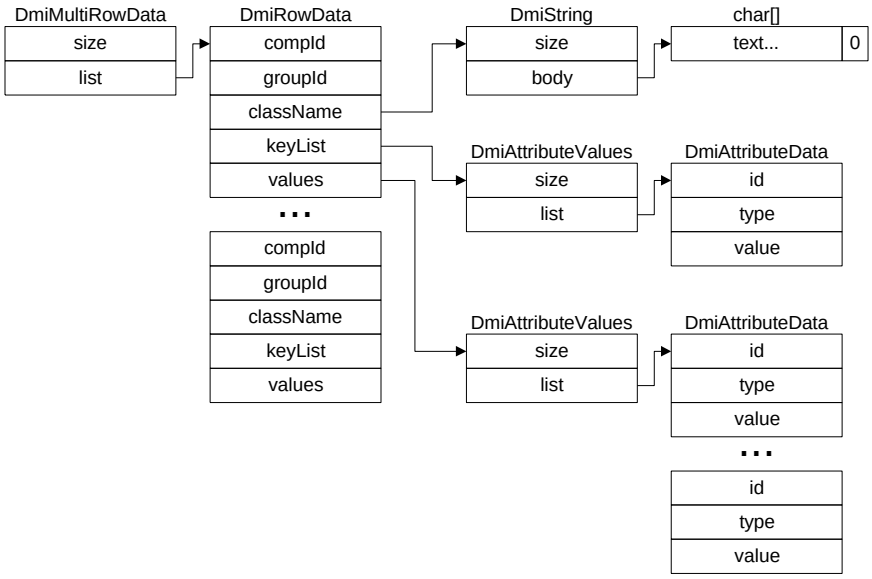
This data structure defines the DMI octet string representation.

FIELD NAME	DESCRIPTION
size	Number of octets in the string body
body	String contents

```
typedef struct DmiOctetString {
    DmiUnsigned_t          size;
    char*                  body;
} DmiOctetString_t;
```

5.3.22 DmiRowData

This data structure identifies {component, group, row, ids} to set.

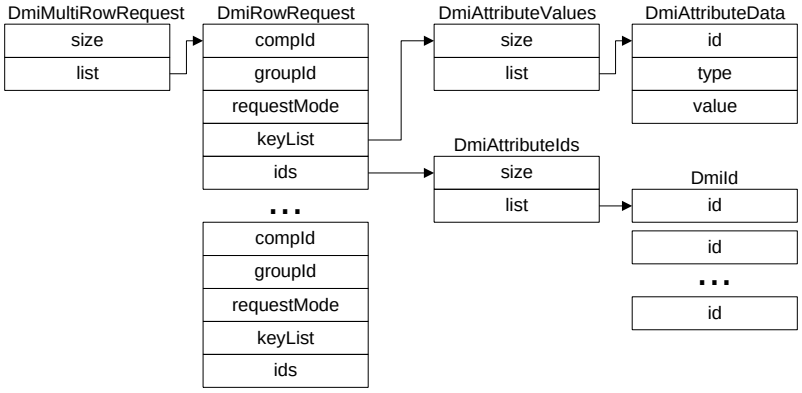


FIELD NAME	DESCRIPTION
compId	Component ID
groupId	Group ID
className	Class name string for the group. Used for indications.
keyList	Array of values for key attributes
values	Array of values for data attributes

```
typedef struct DmiRowData {
    DmiId_t                compId;
    DmiId_t                groupId;
    DmiString_t*           className;
    struct DmiAttributeValues*     keyList;
    struct DmiAttributeValues*     values;
} DmiRowData_t;
```

5.3.23 DmiRowRequest

This data structure identifies {component, group, row, ids} to get.



FIELD NAME	DESCRIPTION
compId	Component ID
groupId	Group ID
requestMode	Get from specified row, first row, or next row
keyList	Array of values for key attributes
ids	Array of Ids for data attributes

```
typedef struct DmiRowRequest {
    DmiId_t                compId;
    DmiId_t                groupId;
    DmiRequestMode_t       requestMode;
    struct DmiAttributeValues*     keyList;
    struct DmiAttributeIds*        ids;
} DmiRowRequest_t;
```

5.3.24 DmiString

This data structure defines the DMI string representation. All DmiStrings must be null terminated. A display string with zero displayable characters still contains the null terminator, and thus has a non-zero length. For the ISO8859-1 character format, the string length for this empty string is 1.

FIELD NAME	DESCRIPTION
size	Number of octets in the string body including the terminating null character (Note: null is 2 octets in Unicode)
body	String contents

```
typedef struct DmiString {
    DmiUnsigned_t          size;
    char*                  body;
} DmiString_t;
```

5.3.25 DmiStringList

This data structure describes a conformant array of DmiString structs.

FIELD NAME	DESCRIPTION
size	Array elements
list	Array data

```
typedef struct DmiStringList {
    DmiUnsigned_t          size;
    DmiString_t**          list;
} DmiStringList_t;
```

5.3.26 DmiTimeStamp

This data structure describes the time format used by DMI. The format of the time block is a 28-octet displayable string with ISO 8859-1 encoding, so each element is one or more printable characters.

For example, Wednesday May 25, 1994 at 1:30:15 PM EDT would be represented as:

```
19940525133015.000000-300
```

A seconds value of 60 is used for leap seconds.

The offset from UTC is the number of minutes west (negative number) or east offset from UTC that indicates the time zone of the system.

Values must be zero-padded if necessary, like "05" in the example above. If a value is not supplied for a field, each character in the field must be replaced with asterisk ('*') characters.

The DMI Server is not required to check the contents of this string for validity.

FIELD NAME	DESCRIPTION
year	The year
month	The month (‘1’..’12’)
day	The day of the month (‘1’..’31’)
hour	The hour of the day (‘0’..’23’)
minutes	The minutes (‘0’..’59’)
seconds	The seconds (‘0’..’60’)
dot	A dot (‘.’}
microseconds	Microseconds (‘0’..’999999’)
plusORminus	‘+’ for east, or ‘-’ west of UTC
utcOffset	Minutes (‘0’..’720’) from UTC
padding	Unused padding for 4-byte alignment

```
typedef struct DmiTimestamp {
    char      year         [4];
    char      month        [2];
    char      day          [2];
    char      hour         [2];
    char      minutes      [2];
    char      seconds      [2];
    char      dot;
    char      microseconds [6];
    char      plusORminus;
    char      utcOffset    [3];
    char      padding      [3];
} DmiTimeStamp_t;
```

6. MANAGEMENT INTERFACE

The functions that comprise the Management Interface (MI) belong to the API described as the ***Service Provider API for Management Applications***. Please see Section 4 for a discussion of the abstract classes of interfaces in the DMI. Also see Section 4.1 for a description of explicit versus implicit bindings. If the Service Provider implements the DMI Security Extension, Management Interface calls are authorized as described in section 13.

6.1 INITIALIZATION FUNCTIONS

DMIv2.0 retains the concept of registration of management applications to the DMI Service Provider agent. The functions DmiRegister and DmiUnregister provide this capability. Some of the data carried in each command in DMIv1.x DmiMgmtCommand block has been extracted. This information is set with a DmiSetConfig call and accessed by DmiGetConfig. These calls contain fields which rarely change between a manager and a client. DmiGetVersion is pulled out as a separate call rather than being a byproduct of the DmiRegisterMgmtReq as it was in DMIv1.x.

6.1.1 DmiRegister

The DmiRegister procedure provides the management application with a unique per-session handle. The DMI Service Provider uses this procedure to initialize its internal state for subsequent procedure calls made by the application. This must be the first DMI command executed by the application. Upon registration, the DMIv2.0s Service Provider associates the roles of the management application user with the allocated management handle, as described in section 13.1.

PARAMETER NAME	DIRECTION	DESCRIPTION
handle	Out	On completion, an open session handle

```
DmiErrorStatus_t DMI_API
DmiRegister (
    [out]    DmiHandle_t*        handle );
```

The client provides the address of the handle parameter and the server fills it in. All commands except DmiRegister() require a valid handle, so this must be the first command sent to the server.

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
```

6.1.2 DmiUnregister

The DmiUnregister procedure must be the last DMI command executed by the management application. The DMI Service Provider uses this procedure to perform its end-of-session cleanup actions. On return from this function, the session handle is no longer valid.

PARAMETER NAME	DIRECTION	DESCRIPTION
handle	In	An open session handle to be closed

```
DmiErrorStatus_t DMI_API
DmiUnregister (
    [in]     DmiHandle_t  handle );
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
```

6.1.3 DmiGetVersion

The DmiGetVersion procedure retrieves information about the DMI Service Provider. The management application uses this procedure to determine the DMI specification level supported by the service provider. This procedure also returns the service provider description string, and may contain version information about the service provider implementation.

PARAMETER NAME	DIRECTION	DESCRIPTION
handle	In	An open session handle
dmiSpecLevel	Out	The DMI Specification version
description	Out	The os-specific DMI Service Provider version
fileTypes	Out	The file types supported for MIF installation

```
DmiErrorStatus_t DMI_API
DmiGetVersion (
    [in]     DmiHandle_t                handle,
    [out]    DmiString_t**              dmiSpecLevel,
    [out]    DmiString_t**              description,
    [out]  DmiFileTypeList_t**          fileTypes );
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
```

6.1.4 DmiGetConfig

The DmiGetConfig procedure retrieves the per-session configuration information. For the DMIv2.0 specification, this configuration information consists of a string describing the current language in use for the session.

PARAMETER NAME	DIRECTION	DESCRIPTION
handle	In	An open session handle
language	Out	language-code | territory-code | encoding

```
DmiErrorStatus_t DMI_API
DmiGetConfig (
    [in]     DmiHandle_t                handle,
    [out]    DmiString_t**              language );
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
```

6.1.5 DmiSetConfig

The DmiSetConfig procedure sets the per-session configuration information. For the DMIv2.0 specification, this configuration information consists of a string describing the language required by the management application.

PARAMETER NAME	DIRECTION	DESCRIPTION
handle	In	An open session handle
language	In	language-code | territory-code | encoding

```
DmiErrorStatus_t DMI_API
DmiSetConfig (
    [in]     DmiHandle_t                handle,
    [in]     DmiString_t*               language );
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
DMIERR_ILLEGAL_TO_SET
 DMIERR_DEFAULT_LANGUAGE_RETURNED
```

6.2 LISTING FUNCTIONS

Discovery functions retain the DMIv1.1 model of sequential or random access to the component, group, and attribute information. Each function takes a requestMode parameter, allowing the caller to specify DMI_FIRST, DMI_NEXT, or DMI_UNIQUE when accessing the information.

In addition, the component list commands have been separated into individual calls to retrieve group classes within a component, to use filtering options, and to retrive mapping files.

Note: commands that allow for the retrieval of pragma or description strings will return a NULL pointer if the string is unavailable. This note applies to component, group, and attribute listings.

6.2.1 DmiListComponents

This call retrieves the name and (optionally) the description of components in a system. This command is used to interrogate a system to determine what components are installed. An enumeration can access a specific component or may be used to sequentially access all components in a system. The caller may choose not to retrieve the component description by setting the value *getDescription* to false. The caller may choose not to retrieve the pragma string by setting the value of getPragma to false.

The maxCount, requestMode, and compId parameters allow the caller to control the information returned by the DMI Service Provider. When the requestMode is DMI_UNIQUE, compId specifies the first component requested (or only component if maxCount is one). When the requestMode is DMI_NEXT, compId specifies the component just before the one requested. When requestMode is DMI_FIRST, compId is unused.

To control the amount of information returned, the caller sets maxCount to something other than zero. The service provider must honor this limit on the amount of information returned. When maxCount is zero the service provider returns information for all components, subject to the constraints imposed by requestMode and compId.

PARAMETER NAME	DIRECTION	DESCRIPTION
handle	In	An open session handle
requestMode	In	Unique, first, or next
maxCount	In	Maximum number to return, or 0 for all
getPragma	In	Get optional pragma string ?
getDescription	In	Get optional component description ?
compId	In	Component to start with (see requestMode)
reply	Out	List of components

```
DmiErrorStatus_t DMI_API
DmiListComponents (
    [in]    DmiHandle_t          handle,
    [in]    DmiRequestMode_t            requestMode,
    [in]    DmiUnsigned_t        maxCount,
    [in]    DmiBoolean_t         getPragma,
    [in]    DmiBoolean_t         getDescription,
    [in]    DmiId_t              compId,
    [out]   DmiComponentList_t** reply );
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
```

```
DMIERR_COMPONENT_NOT_FOUND
DMIERR_DATABASE_CORRUPT
DMIERR_FILE_ERROR
DMIERR_DEFAULT_LANGUAGE_RETURNED
```

6.2.2 DmiListComponentsByClass

This command lists components which match specified criteria. This command is used to determine if a component contains a certain group or a certain row in a table. A filter condition may be that a component contains a specified group class name or that it contains a specific row in a specific group. As with DmiListComponents, the description and pragma strings are optional return values.

Also, see DmiListComponents for an explanation of how requestMode, maxCount, and compId interact to select the information returned.

PARAMETER NAME	DIRECTION	DESCRIPTION
handle	In	An open session handle
requestMode	In	Unique, first, or next
maxCount	In	Maximum number to return, or 0 for all
getPragma	In	Get optional pragma string ?
getDescription	In	Get optional component description
compId	In	Component to start with (see requestMode)
className	In	Group class name string to match
keyList	In	Group row keys to match, or null
reply	Out	List of components

```
DmiErrorStatus_t DMI_API
DmiListComponentsByClass (
    [in]     DmiHandle_t          handle,
    [in]     DmiRequestMode_t             requestMode,
    [in]     DmiUnsigned_t        maxCount,
    [in]     DmiBoolean_t         getPragma,
    [in]     DmiBoolean_t         getDescription,
    [in]     DmiId_t              compId,
    [in]     DmiString_t*         className,
    [in]     DmiAttributeValues_t*        keyList,
    [out]    DmiComponentList_t** reply );
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
DMIERR_COMPONENT_NOT_FOUND
DMIERR_NO_DESCRIPTION
DMIERR_NO_PRAGMA
DMIERR_DATABASE_CORRUPT
DMIERR_FILE_ERROR
DMIERR_DEFAULT_LANGUAGE_RETURNED
```

6.2.3 DmiListLanguages

The DmiListLanguages procedure retrieves the set of language mappings installed for the specified component. The maxCount parameter limits the number of strings returned to the caller.

PARAMETER NAME	DIRECTION	DESCRIPTION
handle	In	An open session handle
maxCount	In	Maximum number to return, or 0 for all
compId	In	Component to access
reply	Out	List of language strings

```
DmiErrorStatus_t DMI_API
DmiListLanguages (
    [in]    DmiHandle_t         handle,
    [in]    DmiUnsigned_t       maxCount,
    [in]    DmiId_t             compId,
    [out]   DmiStringList_t**           reply );
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
DMIERR_COMPONENT_NOT_FOUND
DMIERR_DATABASE_CORRUPT
DMIERR_FILE_ERROR
```

6.2.4 DmiListClassNames

The DmiListClassNames procedure retrieves the class name strings for all groups in a component. This allows the management application to easily determine if a component contains a specific group, or groups. The maxCount parameter limits the number of class name strings returned to the caller.

PARAMETER NAME	DIRECTION	DESCRIPTION
handle	In	An open session handle
maxCount	In	Maximum number to return, or 0 for all
compId	In	Component to access
reply	Out	List of class names and group ids

```
DmiErrorStatus_t DMI_API
DmiListClassNames (
    [in]    DmiHandle_t         handle,
    [in]    DmiUnsigned_t       maxCount,
    [in]    DmiId_t             compId,
    [out]   DmiClassNameList_t** reply );
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
DMIERR_COMPONENT_NOT_FOUND
DMIERR_DATABASE_CORRUPT
DMIERR_FILE_ERROR
```

6.2.5 DmiListGroups

This call retrieves a list of groups within a component. This command can access a specific group or may be used to sequentially access all groups in a component. Note that all enumerations of groups occur within the specified component and do not span components.

The caller may choose not to retrieve the group description by setting the value *getDescription* to false. The caller may choose not to retrieve the pragma string by setting the value of getPragma to false.

The maxCount, requestMode, and groupId parameters allow the caller to control the information returned by the DMI Service Provider. When the requestMode is DMI_UNIQUE, groupId specifies the first group requested (or only group if maxCount is one). When the requestMode is DMI_NEXT, groupId specifies the group just before the one requested. When requestMode is DMI_FIRST, groupId is unused.

To control the amount of information returned, the caller sets maxCount to something other than zero. The service provider must honor this limit on the amount of information returned. When maxCount is zero the service provider returns information for all groups, subject to the constraints imposed by requestMode and groupId.

PARAMETER NAME	DIRECTION	DESCRIPTION
handle	In	An open session handle
requestMode	In	Unique, first, or next group
maxCount	In	Maximum number to return, or 0 for all
getPragma	In	Get optional pragma string ?
getDescription	In	Get optional group description ?
compId	In	Component to access
groupId	In	Group to start with (see requestMode)
reply	Out	List of groups

```
DmiErrorStatus_t DMI_API
DmiListGroups(
    [in]    DmiHandle_t         handle,
    [in]    DmiRequestMode_t            requestMode,
    [in]    DmiUnsigned_t       maxCount,
    [in]    DmiBoolean_t        getPragma,
    [in]    DmiBoolean_t        getDescription,
    [in]    DmiId_t             compId,
    [in]    DmiId_t             groupId,
    [out]   DmiGroupList_t**            reply );
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
DMIERR_COMPONENT_NOT_FOUND
DMIERR_GROUP_NOT_FOUND
DMIERR_NO_PRAGMA
DMIERR_NO_DESCRIPTION
DMIERR_DATABASE_CORRUPT
DMIERR_FILE_ERROR
DMIERR_DEFAULT_LANGUAGE_RETURNED
```

6.2.6 DmiListAttributes

This DmiListAttributes procedure retrieves the properties for one or more attributes in a group. Note that all enumerations of attributes occur within the specified group, and do not span groups.

The caller may choose not to retrieve the description string by setting the value of getDescription to false. Likewise, the caller may choose not to retrieve the pragma string by setting the value of getPragma to false.

The maxCount, requestMode, and attribId parameters allow the caller to control the information returned by the DMI Service Provider. When the requestMode is DMI_UNIQUE, attribId specifies the first attribute requested (or only attribute if maxCount is one). When the requestMode is DMI_NEXT, attribId specifies the attribute just before the one requested. When requestMode is DMI_FIRST, attribId is unused.

To control the amount of information returned, the caller sets maxCount to something other than zero. The service provider must honor this limit on the amount of information returned. When maxCount is zero the service provider returns information for all attributes, subject to the constraints imposed by requestMode and attribId.

P/ RAMETER NAME	)IRECTION	DESCRI 'TION
handle	In	An open session handle
requestMode	In	Unique, first, or next attribute
maxCount	In	Maximum number to return, or 0 for all
getPragma	In	Get optional pragma string ?
getDescription	In	Get optional attribute description ?
compId	In	Component to access
groupId	In	Group to access
attribId	In	Attribute to start with (see requestMode)
reply	Out	List of attributes

```
DmiErrorStatus_t DMI_API
DmiListAttributes(
    [in]     DmiHandle_t                handle,
    [in]     DmiRequestMode_t           requestMode,
    [in]     DmiUnsigned_t              maxCount,
    [in]     DmiBoolean_t               getPragma,
    [in]     DmiBoolean_t               getDescription,
    [in]     DmiId_t                    compId,
    [in]     DmiId_t                    groupId,
```

```
[in]     DmiId_t                          attribId,
[out]    DmiAttributeList_t** reply );
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
DMIERR_ATTRIBUTE_NOT_FOUND
DMIERR_COMPONENT_NOT_FOUND
DMIERR_GROUP_NOT_FOUND
DMIERR_NO_PRAGMA
DMIERR_NO_DESCRIPTION
DMIERR_DATABASE_CORRUPT
DMIERR_FILE_ERROR
DMIERR_DEFAULT_LANGUAGE_RETURNED
```

6.3 OPERATION FUNCTIONS

6.3.1 DmiGetAttribute

The DmiGetAttribute procedure provides a simple method for retrieving a single attribute value from the DMI Service Provider. The compId, groupId, attribId, and keyList identify the desired attribute. The resulting attribute value is returned in a newly allocated DmiDataUnion structure. The address of this structure is returned through the value parameter.

A management application may or may not specify a keylist. When a keylist is omitted for a table access, the Service Provider or instrumentation shall operate on the first row of the table, regardless of the Access Mode specified.

Note: the "first row" of a table will remain constant during the execution of the Service Provider. This is true for both instrumented and non-instrumented tables. The "first row" *can* change between reboots of the system, or restarts of the Service Provider. This restriction ensures that management applications dealing with the first row of a table are always operating on the same row.

PARAMETER NAME	DIRECTION	DESCRIPTION
Handle	In	An open session handle
CompId	In	Component to access
GroupId	In	Group within component
attribId	In	Attribute within group
keyList	In	Keylist to specify a table row
value	Out	Attribute value returned

```
DmiErrorStatus_t DMI_API
DmiGetAttribute (
    [in]      DmiHandle_t            handle,
    [in]      DmiId_t                compId,
    [in]      DmiId_t                groupId,
    [in]      DmiId_t                attribId,
    [in]      DmiAttributeValues_t*  keyList,
    [out]     DmiDataUnion_t**       value );
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
DMIERR_ATTRIBUTE_NOT_FOUND
DMIERR_COMPONENT_NOT_FOUND
DMIERR_GROUP_NOT_FOUND
DMIERR_ILLEGAL_KEYS
DMIERR_OVERLAY_NAME_NOT_FOUND
DMIERR_ILLEGAL_TO_GET
DMIERR_ROW_NOT_FOUND
DMIERR_DIRECT_INTERFACE_NOT_REGISTERED
DMIERR_DATABASE_CORRUPT
DMIERR_ATTRIBUTE_NOT_SUPPORTED
DMIERR_UNKNOWN_CI_REGISTRY
DMIERR_FILE_ERROR
DMIERR_OVERLAY_NOT_FOUND
DMIERR_VALUE_UNKNOWN
```

6.3.2 DmiSetAttribute

The DmiSetAttribute procedure provides a simple method for setting a single attribute value. The compId, groupId, attribId, and keyList identify the desired attribute; the setMode parameter defines the procedure call as a Set, Reserve, or Release operation. The new attribute value is contained in the DmiDataUnion structure whose address is passed in the value parameter.

A management application may or may not specify a keylist. When a keylist is omitted for a table access, the Service Provider or instrumentation shall operate on the first row of the table, regardless of the Access Mode specified.

Note: the "first row" of a table will remain constant during the execution of the Service Provider. This is true for both instrumented and non-instrumented tables. The "first row" *can* change between reboots of the system, or restarts of the Service Provider. This restriction ensures that management applications dealing with the first row of a table are always operating on the same row.

PARAMETER NAME	DIRECTION	DESCRIPTION
handle	In	An open session handle
compId	In	Component to access
groupId	In	Group within component
attribId	In	Attribute within group
keyList	In	Keylist to specify a table row
setMode	In	Set, reserve, or release ?
value	In	Attribute value to set

```
DmiErrorStatus_t  DMI_API
DmiSetAttribute (
    [in]      DmiHandle_t            handle,
    [in]      DmiId_t                compId,
    [in]      DmiId_t                groupId,
    [in]      DmiId_t                attribId,
    [in]      DmiAttributeValues_t*  keyList,
    [in]      DmiSetMode_t           setMode,
    [in]      DmiDataUnion_t*        value );
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
DMIERR_ATTRIBUTE_NOT_FOUND
DMIERR_COMPONENT_NOT_FOUND
DMIERR_GROUP_NOT_FOUND
DMIERR_ILLEGAL_KEYS
DMIERR_OVERLAY_NAME_NOT_FOUND
DMIERR_ILLEGAL_TO_GET
DMIERR_ROW_NOT_FOUND
DMIERR_DIRECT_INTERFACE_NOT_REGISTERED
DMIERR_DATABASE_CORRUPT
DMIERR_ATTRIBUTE_NOT_SUPPORTED
DMIERR_UNKNOWN_CI_REGISTRY
DMIERR_FILE_ERROR
DMIERR_OVERLAY_NOT_FOUND
DMIERR_VALUE_UNKNOWN
```

6.3.3 DmiGetMultiple

The DmiGetMultiple procedure retrieves attribute values from the DMI Service Provider. This command may get the value for an individual attribute, or for multiple attributes across groups, components, or rows of a table.

The request array, described in Section 5.3.16, specifies the attribute values requested by the management application. Each element of the array specifies a component, group, request mode, key list (for table accesses), and attribute list to retrieve. The key list is omitted (NULL pointer value) for scalar groups. If the attribute list is omitted, the service provider returns all attributes in the group or table row. The requestMode specifier allows the management application to request the first, next, or specific attribute value.

The rowData array, described in Sections 5.3.15, contains the reply from the DMI Service Provider. The structure of this reply is identical to that of the original request, with the same number of elements that were in the request array.

A management application may or may not specify a keylist. When a keylist is omitted for a table access, the Service Provider or instrumentation shall operate on the first row of the table, regardless of the Access Mode specified.

Note: the "first row" of a table will remain constant during the execution of the Service Provider. This is true for both instrumented and non-instrumented tables. The "first row" *can* change between reboots of the system, or restarts of the Service Provider. This restriction ensures that management applications dealing with the first row of a table are always operating on the same row.

When DmiGetMultiple is called without an attribute list, the Service Provider returns all attributes in the group or table row. Attributes that are UNSUPPORTED or WRITE-ONLY are omitted from the reply data, and the return status for the operation is DMIERR_NO_ERROR.

When DmiGetMultiple is called with a specific attribute list, the Service Provider returns a value for each requested attribute. Attributes that are UNSUPPORTED or WRITE-ONLY cause the Service Provider to stop processing the request and return data for all attributes up to, but not including, the error attribute.

If partial attribute data is returned, the operation's return status is DMIERR_NO_ERROR_MORE_DATA. When DmiGetMultiple returns a status of DMIERR_NO_ERROR_MORE_DATA, the caller should reissue the operation with a new attribute list. This new attribute list should start with the first attribute *not* returned in the previous call, and should contain all subsequent attributes from the original list.

If the first attribute in the attribute list is UNSUPPORTED, the Service Provider shall stop processing the request and return an error status of DMIERR_ATTRIBUTE_NOT_SUPPORTED.

If the first attribute in the attribute list is WRITE-ONLY, the Service Provider shall stop processing the request and return an error status of DMIERR_ILLEGAL_TO_GET.

PARAMETER NAME	DIRECTION	DESCRIPTION
handle	In	An open session handle
request	In	Attributes to get
rowData	Out	Requested attribute values

```
DmiErrorStatus_t DMI_API
DmiGetMultiple (
    [in]    DmiHandle_t          handle,
    [in]    DmiMultiRowRequest_t*       request,
    [out]   DmiMultiRowData_t**         rowData );
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
DMIERR_ATTRIBUTE_NOT_FOUND
DMIERR_COMPONENT_NOT_FOUND
DMIERR_GROUP_NOT_FOUND
DMIERR_ILLEGAL_KEYS
DMIERR_OVERLAY_NAME_NOT_FOUND
DMIERR_ILLEGAL_TO_GET
DMIERR_ROW_NOT_FOUND
DMIERR_DIRECT_INTERFACE_NOT_REGISTERED
DMIERR_DATABASE_CORRUPT
DMIERR_ATTRIBUTE_NOT_SUPPORTED
DMIERR_UNKNOWN_CI_REGISTRY
DMIERR_FILE_ERROR
DMIERR_OVERLAY_NOT_FOUND
DMIERR_VALUE_UNKNOWN
```

6.3.4 DmiSetMultiple

This command performs a set operation on an attribute or list of attributes. Set operations include actually setting the value, testing and reserving the attribute for future setting, or releasing the set reserve. These variations on the set operation are specified by the parameter setMode.

The rowData array describes the attributes to set, and contains the new attribute values. Each element of rowData specifies a component, group, key list (for table accesses), and attribute list to set. No data is returned from this function.

A management application may or may not specify a keylist. When a keylist is omitted for a table access, the Service Provider or instrumentation shall operate on the first row of the table, regardless of the Access Mode specified.

Note: the "first row" of a table will remain constant during the execution of the Service Provider. This is true for both instrumented and non-instrumented tables. The "first row" *can* change between reboots of the system, or restarts of the Service Provider. This restriction ensures that management applications dealing with the first row of a table are always operating on the same row.

PARAMETER NAME	DIRECTION	DESCRIPTION
handle	In	An open session handle
setMode	In	Set, reserve, or release
rowData	In	Attribute values to set

```
DmiSetMultiple (
    [in]     DmiHandle_t                handle,
    [in]     DmiSetMode_t               setMode,
    [in]     DmiMultiRowData_t*         rowData );
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
DMIERR_ATTRIBUTE_NOT_FOUND
DMIERR_VALUE_EXCEEDS_MAXSIZE
DMIERR_COMPONENT_NOT_FOUND
DMIERR_GROUP_NOT_FOUND
DMIERR_ILLEGAL_KEYS
DMIERR_ILLEGAL_TO_SET
DMIERR_OVERLAY_NAME_NOT_FOUND
```

```
DMIERR_ROW_NOT_FOUND
DMIERR_DIRECT_INTERFACE_NOT_REGISTERED
DMIERR_DATABASE_CORRUPT
DMIERR_ATTRIBUTE_NOT_SUPPORTED
DMIERR_UNKNOWN_CI_REGISTRY
DMIERR_FILE_ERROR
DMIERR_OVERLAY_NOT_FOUND
DMIERR_VALUE_UNKNOWN
```

6.3.5 DmiAddRow

The DmiAddRow procedure adds a row to an existing table. The rowData parameter contains the full data, including key attribute values, for a row. It is an error for the key list to specify an existing table row.

When a table contains a mix of instrumented and non-instrumented attributes, the DmiAddRow operation is not permitted. This restriction is necessary because the Service Provider does not know whether to add the row in the MIF database, or in the (partially) supporting instrumentation. The Service Provider will fail the operation with a DMIERR_UNABLE_TO_ADD_ROW status.

Note that, from both a design and implementation standpoint, it is generally a bad idea to mix instrumented and non-instrumented values in a table. This is especially true where keys are concerned. Synchronization between the component attributes and database attributes is problematic, at best. A case where some keys reside in component instrumentation and other keys reside in the MIF database is nearly impossible to implement in the Service Provider, or manage in component instrumentation. It is **STRONGLY** recommended that component providers do **NOT** mix table rows in this way.

PARAMETER NAME	DIRECTION	DESCRIPTION
handle	In	An open session handle
rowData	In	Attribute values to set

```
DmiErrorStatus_t DMI_API
DmiAddRow (
    [in]     DmiHandle_t         handle,
    [in]     DmiRowData_t*       rowData );
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
DMIERR_VALUE_EXCEEDS_MAXSIZE
DMIERR_COMPONENT_NOT_FOUND
DMIERR_GROUP_NOT_FOUND
DMIERR_ILLEGAL_KEYS
DMIERR_OVERLAY_NAME_NOT_FOUND
DMIERR_ROW_NOT_FOUND
DMIERR_DIRECT_INTERFACE_NOT_REGISTERED
DMIERR_DATABASE_CORRUPT
DMIERR_ATTRIBUTE_NOT_SUPPORTED
DMIERR_UNKNOWN_CI_REGISTRY
DMIERR_FILE_ERROR
DMIERR_OVERLAY_NOT_FOUND
DMIERR_VALUE_UNKNOWN
DMIERR_UNABLE_TO_ADD_ROW
```

6.3.6 DmiDeleteRow

The DmiDeleteRow procedure removes a row from an existing table. The key list must specify valid keys for a table row.

PARAMETER NAME	DIRECTION	DESCRIPTION
handle	In	An open session handle
rowData	In	Row to delete

```
DmiErrorStatus_t DMI_API
DmiDeleteRow (
    [in]     DmiHandle_t         handle,
    [in]     DmiRowData_t*       rowData );
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
DMIERR_ATTRIBUTE_NOT_FOUND
DMIERR_COMPONENT_NOT_FOUND
DMIERR_GROUP_NOT_FOUND
DMIERR_ILLEGAL_KEYS
DMIERR_OVERLAY_NAME_NOT_FOUND
DMIERR_ILLEGAL_TO_GET
DMIERR_ROW_NOT_FOUND
DMIERR_DIRECT_INTERFACE_NOT_REGISTERED
DMIERR_DATABASE_CORRUPT
DMIERR_ATTRIBUTE_NOT_SUPPORTED
DMIERR_UNKNOWN_CI_REGISTRY
DMIERR_FILE_ERROR
DMIERR_OVERLAY_NOT_FOUND
DMIERR_VALUE_UNKNOWN
DMIERR_UNABLE_TO_DELETE_ROW
```

6.4 DATABASE ADMINISTRATION FUNCTIONS

The APIs listed in this section modify the schema of the database.

6.4.1 DmiAddComponent

The DmiAddComponent procedure is used to add a new component to the DMI database. It takes the name of a file, or the address of memory block containing schema description data, checks the data for adherence to the appropriate schema description format (e.g. DMI MIF format), and installs the schema description in the database. The procedure returns a unique component ID for the newly installed component.

PARAMETER NAME	DIRECTION	DESCRIPTION
handle	In	An open session handle
fileData	In	Schema description file data for the component
compId	Out	On Completion, the SP-allocated component ID
errors	Out	Installation error messages

```
DmiErrorStatus_t DMI_API
DmiAddComponent()
    [in]     DmiHandle_t          handle,
    [in]     DmiFileDataList_t*           fileData,
    [out]    DmiId_t*             compId,
    [out]    DmiStringList_t**            errors );
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
DMIERR_DATABASE_CORRUPT
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_FILE_ERROR
DMIERR_BAD_SCHEMA_DESCRIPTION_FILE
DMIERR_INVALID_FILE_TYPE
DMIERR_FILE_TYPE_NOT_SUPPORTED
```

6.4.2 DmiAddLanguage

The DmiAddLanguage procedure is used to add a new language mapping for an existing component in the database. It takes the name of a file, or the address of memory block containing translated schema description data, checks the data for adherence to the schema description grammar (e.g. DMI MIF grammar), and installs the translated schema description in the database.

The description of the new language mapping must match the currently installed component's groups and attributes, excluding names, descriptions, pragmas, and values. That is, the structure of the component must be maintained by the new language mapping.

PARAMETER NAME	DIRECTION	DESCRIPTION
handle	In	An open session handle
fileData	In	Language mapping file for the component
compId	In	Component to access
errors	Out	Installation error messages

```
DmiErrorStatus_t DMI_API
DmiAddLanguage (
    [in]     DmiHandle_t          handle,
    [in]     DmiFileDataList_t*           fileData,
    [in]     DmiId_t              compId,
    [out]    DmiStringList_t**            errors );
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
DMIERR_COMPONENT_NOT_FOUND
DMIERR_DATABASE_CORRUPT
DMIERR_FILE_ERROR
DMIERR_BAD_SCHEMA_DESCRIPTION_FILE
DMIERR_INVALID_FILE_TYPE
DMIERR_FILE_TYPE_NOT_SUPPORTED
```

6.4.3 DmiAddGroup

The DmiAddGroup procedure is used to add a new group to an existing component in the database. It takes the name of a file, or the address of memory block containing the group's schema description data, checks the data for adherence to the schema description grammar (e.g. DMI MIF grammar), and installs the group schema description in the database.

When the DmiFileType is DMI_GROUP_FILE_NAME or DMI_GROUP_FILE_DATA, the format of the data must be a valid component definition containing a single group definition. This means that the data must include both START COMPONENT and END COMPONENT declarations, and may include, for example, PATH statements and ENUM definitions at the component level.

Note that certains restrictions apply to the schema supplied for DmiAddGroup():

- Table Definitions are disallowed
- One and only one Group Definition is allowed. This group definition MUST specify a group ID (i.e., it may not be an uninstantiated template).

Schema violating these restrictions will be rejected by the Service Provider with a status of DMIERR_BAD_SCHEMA_DESCRIPTION_FILE.

When adding a group to component that already has multiple languages installed, the fileData included with DmiAddGroup must contain a group definition for each installed language. Tthis ensures that a *complete* language mapping is always available for a component.

PARAMETER NAME	DIRECTION	DESCRIPTION
handle	In	An open session handle
fileData	In	Schema description file data for the group definition
compId	In	Component to access
groupId	Out	On completion, the SP-allocated group ID
errors	Out	Installation error messages

```
DmiErrorStatus_t DMI_API
DmiAddGroup (
    [in]     DmiHandle_t          handle,
    [in]     DmiFileDataList_t*           fileData,
    [in]     DmiId_t              compId,
    [out]    DmiId_t              groupId,
    [out]    DmiStringList_t**            errors );
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
DMIERR_COMPONENT_NOT_FOUND
DMIERR_DATABASE_CORRUPT
DMIERR_FILE_ERROR
DMIERR_BAD_SCHEMA_DESCRIPTION_FILE
DMIERR_INVALID_FILE_TYPE
```

6.4.4 DmiDeleteComponent

The DmiDeleteComponent procedure is used to remove an existing component from the database.

PARAMETER NAME	DIRECTION	DESCRIPTION
handle	In	An open session handle
compId	In	Component to delete

```
DmiErrorStatus_t DMI_API
DmiDeleteComponent (
    [in]     DmiHandle_t          handle,
    [in]     DmiId_t              compId );
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
DMIERR_COMPONENT_NOT_FOUND
DMIERR_DATABASE_CORRUPT
DMIERR_FILE_ERROR
DMIERR_CANT_UNINSTALL_SP_COMPONENT
```

6.4.5 DmiDeleteLanguage

The DmiDeleteLanguage procedure is used to remove a specific language mapping for a component. The caller specifies the language string and component ID.

PARAMETER NAME	DIRECTION	DESCRIPTION
handle	In	An open session handle
language	In	language-code | territory-code | encoding
compId	In	Component to access

```
DmiErrorStatus_t DMI_API
DmiDeleteLanguage (
    [in]     DmiHandle_t         handle,
    [in]     DmiString_t*        language,
    [in]     DmiId_t             compId );
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
DMIERR_COMPONENT_NOT_FOUND
DMIERR_DATABASE_CORRUPT
DMIERR_FILE_ERROR
DMIERR_CANT_UNINSTALL_COMPONENT_LANGUAGE
```

6.4.6 DmiDeleteGroup

The DmiDeleteGroup procedure is used to remove a group from a component. The caller specifies the component and group IDs.

PARAMETER NAME	DIRECTION	DESCRIPTION
handle	In	An open session handle
compId	In	Component containing group
groupId	In	Group to delete

```
DmiErrorStatus_t DMI_API
DmiDeleteGroup (
    [in]     DmiHandle_t         handle,
    [in]     DmiId_t             compId,
    [in]     DmiId_t             groupId );
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
DMIERR_COMPONENT_NOT_FOUND
DMIERR_GROUP_NOT_FOUND
DMIERR_DATABASE_CORRUPT
DMIERR_FILE_ERROR
DMIERR_CANT_UNINSTALL_GROUP
```

7. MANAGEMENT APPLICATION PROVIDER API

7.1 FUNCTIONS

This section describes the functions that a client must provide to receive indications. These functions belong to the API described as the ***Management Application Provider Functions***. Please see Section 4 for a discussion of the abstract classes of interfaces in the DMI.

A client receiving indications undergoes a role reversal where, in RPC terms, it becomes an indication delivery server. The DMI Service Provider is a client of this interface.

There are eight indication types defined by the DMTF: add/delete component, add/delete language mapping, add/delete group, subscription expiration notice, and event delivery. Each indication arrives at a unique entry point in the indication interface.

All indication functions have some information in common, and some that is unique to the indication. The first piece of common information is the opaque handle returned to the application. This handle contains the SubscriberID attribute from the client's row in the SPIndicationSubscription table. This can be used by the indication delivery interface to determine which local management application should receive the indication.

The second piece of common information is the sender's address. Since indications can arrive from any number of remote systems, the receiver needs a way to determine its origin. The sender's address provides this mechanism.

The eight entry points, including their specific details, are described in the following sections.

7.1.1 DmiDeliverEvent

This command delivers event data to an application.

PARAMETER NAME	DIRECTION	DESCRIPTION
handle	In	An opaque ID returned to the
sender	In	Address of the node delivering the
language	In	Language encoding for the indication
compId	In	Component reporting the event
timestamp	In	Event generation time
rowData	In	Standard and context-specific indication

```
DmiDeliverEvent (
    [in]    DmiUnsigned_t        handle,
    [in]    DmiNodeAddress_t*           sender,
    [in]    DmiString_t*         language,
    [in]    DmiId_t              compId,
    [in]    DmiTimestamp_t*             timestamp,
    [in]    DmiMultiRowData_t*          rowData );
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
```

7.1.2 DmiComponentAdded

PARAMETER NAME	DIRECTION	DESCRIPTION
handle	In	An opaque ID returned to the application
sender	In	Address of the node delivering the indication
info	In	Information about the component added

```
DmiErrorStatus_t DMI_API
DmiComponentAdded (
    [in]  DmiUnsigned_t           handle,
    [in]  DmiNodeAddress_t*              sender,
    [in]  DmiComponentInfo_t*            info );
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
```

7.1.3 DmiComponentDeleted

PARAMETER NAME	DIRECTION	DESCRIPTION
handle	In	An opaque ID returned to the application
sender	In	Address of the node delivering the indication
compId	In	Component deleted from the data base

```
DmiErrorStatus_t DMI_API
DmiComponentDeleted (
    [in]  DmiUnsigned_t           handle,
    [in]  DmiNodeAddress_t*              sender,
    [in]  DmiId_t                 compId );
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
```

```
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
```

7.1.4 DmiLanguageAdded

PARAMETER NAME	DIRECTION	DESCRIPTION
handle	In	An opaque ID returned to the application
sender	In	Address of the node delivering the indication
compId	In	Component with new language mapping
language	In	Language-code | territory-code | encoding

```
DmiErrorStatus_t DMI_API
DmiLanguageAdded (
    [in]  DmiUnsigned_t           handle,
    [in]  DmiNodeAddress_t*              sender,
    [in]  DmiId_t                 compId,
    [in]  DmiString_t*            language );
```

ERROR CODES

```
DmiLanguageAdded(handle,sender,compid,language)
    DMIERR_NO_ERROR
    DMIERR_ILLEGAL_HANDLE
    DMIERR_OUT_OF_MEMORY
    DMIERR_INSUFFICIENT_PRIVILEGES
    DMIERR_SP_INACTIVE
```

7.1.5 DmiLanguageDeleted

PARAMETER NAME	DIRECTION	DESCRIPTION
handle	In	An opaque ID returned to the application
sender	In	Address of the node delivering the indication
compId	In	Component with deleted language mapping
language	In	Language-code | territory-code | encoding

```
DmiErrorStatus_t DMI_API
DmiLanguageDeleted (
    [in]  DmiUnsigned_t           handle,
    [in]  DmiNodeAddress_t*              sender,
    [in]  DmiId_t                 compId,
    [in]  DmiString_t*            language );
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
```

7.1.6 DmiGroupAdded

PARAMETER NAME	DIRECTION	DESCRIPTION
handle	In	An opaque ID returned to the application
sender	In	Address of the node delivering the indication
compId	In	Component with new group added
info	In	Information about the new group added

```
DmiErrorStatus_t DMI_API
DmiGroupAdded (
    [in]  DmiUnsigned_t           handle,
    [in]  DmiNodeAddress_t*               sender,
    [in]  DmiId_t                 compId,
    [in]  DmiGroupInfo_t*         info );
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
```

7.1.7 DmiGroupDeleted

PARAMETER NAME	DIRECTION	DESCRIPTION
handle	In	An opaque ID returned to the application
Sender	In	Address of the node delivering the indication
CompId	In	Component with the group deleted
GroupId	In	Group deleted from the component

```
DmiErrorStatus_t DMI_API
DmiGroupDeleted (
    [in]  DmiUnsigned_t           handle,
    [in]  DmiNodeAddress_t*               sender,
    [in]  DmiId_t                 compId,
    [in]  DmiId_t                 groupId );
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
```

7.1.8 DmiSubscriptionNotice

In order to receive indications, a managing node must have subscribed for indications with a managed node. The process for doing this is basically the populating of a row in the SPIndicationSubscription table on the managed node. This can be accomplished using the DmiAddRow() and DmiDeleteRow() commands defined elsewhere in this document. Among the attributes in this group, are an expiration date for this subscription, and a date on which the service provider should start warning the managing node of a pending expiration. The DMI Service Provider is responsible for sending two types of indications to the managing node, based on these dates, to inform it that its current subscription is either about to expire, or has expired, and it does so using DmiSubscriptionNotice.

NOTE: for a complete description of how the managed node determines when to send the expiration pending indication, see the group definition for "SPIndicationSubscription", Section 3.3.1.

PARAMETER NAME	I IRECTION	DESCRIPTION
handle	In	An opaque ID returned to the application
sender	In	Address of the originating node
expired	In	False: Subscription expiration pending True: Subscription has expired
rowData	In	Information about this subscription. This will be the row information for the appropriate entry in the indication table defined by the "SPIndicationSubscription" group.

```
DmiErrorStatus_t  DMI_API
DmiSubscriptionNotice (
    [in]  DmiUnsigned_t    handle,
    [in]  DmiNodeAddress_t*       sender,
    [in]  DmiBoolean_t     expired,
    [in]  DmiRowData_t*    rowData );
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
```

8. COMPONENT INTERFACE

The Component Interface (CI) is an optional interface allowing managed components to connect directly to the DMI Service Provider. Note that the capabilities provided by this interface are often platform or operating system specific. For this reason the Distributed Management Task Force, the administrative body responsible for the DMI, has made the CI optional and therefore not a requirement for an implementation to be considered conformant to the DMI model. It is included here for continuity from the DMIv1.1 Specification (hereafter referred to as DMIv1.x).

In the DMIv1.x, the CI provides calls necessary for a managed component to install/uninstall with the DMI Service Provider. In the procedural DMI model, equivalent functionality is provided by add/delete component calls across the remotable MI layer.

The DMIv1.x CI model uses 'well known entry points' *DmiCiInvoke()* and *DmiCiCancel()* to set up and cancel commands destined for CI instrumentation. These entry points are no longer needed as this functionality will be handled within the DMI Service Provider. Instead, the procedural CI will make use of entry points to five well known procedures common to DMIv1.x instrumentation: *ciGetAttribute(), ciGetNextAttribute, ciReserveAttribute(), ciSetAttribute*, and *ciReleaseAttribute()*. Two new entry points are added for manipulating instrumented tables: ciAddRow() and ciDeleteRow().

The procedural CI uses formalized data structures instead of block oriented commands as in DMIv1.x. The interface is completely synchronous with the service provider acting as the broker to ensure that component code need not be re-entrant.

DMIv2.0s defines two features of the Component Interface: allowing only privileged processes to register component instrumentation and disabling of component instrumentation override. These features are described in section 14.

8.1 DATA STRUCTURES

8.1.1 DmiAccessData

This data structure contains group/attribute access ID for instrumentation wishing to register for the direct interface.

FIELD NAME	DESCRIPTION
GroupId	Group that uses the direct interface. A value of zero indicates that all groups within this MIF use the direct interface, and the following iAttributeId field is ignored.
attributeId	Attributes, within the group specified by GroupId, that use the direct interface. A value of zero indicates that all attributes within this group use the direct interface.

```
typedef struct DmiAccessData {
    DmiId_t   groupId;
    DmiId_t   attributeId;
 } DmiAccessData_t;
```

8.1.2 DmiAccessDataList

This data structure contains describes an array of DmiAccessData structs.

FIELD NAME	DESCRIPTION
size	Array elements
List	Array data

```
typedef struct DmiAccessData {
    DmiUnsigned_t          size;
    DmiAccessData_t*       list;
 } DmiAccessDataList_t;
```

8.1.3 DmiRegisterInfo

This data structure identifies entry points for registering CI direct interface code.

FIELD NAME	DESCRIPTION
componentId	Identifier assigned by the service provider on component installation
ciGetAttribute	Address ot the CiGetAttribute entry point
ciGetNextAttribute	Address of the CiGetNextAttribute entry point
ciReserveAtttribute	Address of the CiReserveAttrribute entry point
ciReleaseAtttribute	Address of the CiReleaseAttrribute entry point
ciSetAttribute	Address of the CiSetAttribute entry point
ciAddRow	Address of the CiAddRow entry point
ciDeleteRow	Address of the CiDeleteRow entry point
accessData	Array containing the groups and/or individual attributes that use the direct interface

```
typedef struct DmiRegisterInfo {
    DmiId_t                       componentId;
    CiGetAttribute*               ciGetAttribute;
    CiGetNextAttribute*           ciGetNextAttribute;
    CiReserveAttribute*           ciReserveAttribute;
    CiReleaseAttribute*           ciReleaseAttribute;
    CiSetAttribute*               ciSetAttribute;
    CiAddRow*                     ciAddRow;
    CiDeleteRow*                  ciDeleteRow;
    DmiAccessDataList_t*          accessData;
}DmiRegisterInfo_t;
```

8.2 SERVICE PROVIDER FUNCTIONS FOR COMPONENTS

The functions described in this section belong to the API described as the ***Service Provider Functions for Components***. Please see Section 4 for a discussion of the abstract classes of interfaces in the DMI.

In the DMIv1.x block model, the *DmiInvoke()* entry point was called with a DMI command block. *DmiInvoke()* built a CI command block and called *DmiProcess()* to interpret the command and dispatch the appropriate Get and Set operations. Instead, the procedural CI consists of five public entry points in component code called directly from the service provider.

Component instrumentation code may register with the service provider to override its current access mechanism for the registered attributes. Instead of manipulating the data in the MIF database or invoking programs, the service provider will call the entry points provided in the registration call. Once the component unregisters, the SP will return to its "normal method" of processing requests for the data as defined in the MIF. In this way, component instrumentation can temporarily interrupt normal processing to perform some special function. Note that registering attributes through the direct interface will override attributes that are already being served through the direct interface.

8.2.1 DmiRegisterCi Function

The *DmiRegisterCi()* call is used to register a callable interface for components that have resident instrumentation code and/or to get the version of the service provider. Service Providers that implement the DMI Security Extension defined in ***DMIv2.0s*** will check if the caller is a privileged process and if the *DmiRegisterCi()* call would override a previous instrumentation registration, as defined in section 14.

PARAMETER NAME	DIRECTION	DESCRIPTION
regInfo	In	Data structure containing component, group and attribute Ids, as well as pointers to component instrumentation entry points
handle	Out	Service provider assigned handle uniquely identifying this component instrumentation
dmiSpecLevel	Out	The service provider version string

```
DmiErrorStatus_t DMI_API
DmiRegisterCi (
    [in]     DmiRegisterInfo_t*          regInfo,
    [out]    DmiHandle_t*         handle,
    [out]    DmiString_t**        dmiSpecLevel);
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
DMIERR_ATTRIBUTE_NOT_FOUND
DMIERR_COMPONENT_NOT_FOUND
DMIERR_GROUP_NOT_FOUND
DMIERR_DATABASE_CORRUPT
DMIERR_OUT_OF_MEMORY
DMIERR_ILLEGAL_DMI_LEVEL
```

8.2.2 DmiUnregisterCi Function

DmiUnregisterCi() tells the service provider to remove a direct component instrumentation interface from the service provider's table of registered interfaces. This procedural *DmiUnregisterCI()* call is simplified over the DMIv1.x model for unregistering component instrumentation, requiring a single parameter: the service provider assigned handle given to instrumentation at registration time.

PARAMETER NAME	DIRECTION	DESCRIPTION
handle	In	Service provider assigned handle uniquely identifying this component instrumentation

```
DmiErrorStatus_t DMI_API
DmiUnregisterCi (
    [in]     DmiHandle_t  handle);
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
DMIERR_UNKNOWN_CI_REGISTRY
```

8.2.3 DmiOriginateEvent

This function call originates an event for filtering and delivery. Any necessary indication filtering is performed by this function (or by subsequent processing) before the event is forwarded to the management applications. Implementation note: a *compID* value of zero (0) specifies that the event was generated by something that has not been installed as a component, and hence has no component ID.

PARAMETER NAME	DIRECTION	DESCRIPTION
compId	In	Component reporting the event
language	In	language-code | territory-code | encoding
timestamp	In	Event generation time
rowData	In	Standard and context-specific indication data

```
DmiErrorStatus_t DMI_API
DmiOriginateEvent (
    [in]     DmiId_t             compId,
    [in]     DmiString_t*        language,
    [in]     DmiTimestamp_t*     timestamp,
    [in]     DmiMultiRowData_t*  rowData );
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
```

8.3 COMPONENT PROVIDER FUNCTIONS

The functions in this section belong to the API described as the ***Component Provider Functions***. See Section 4 for a discussion of the abstract classes of APIs in the DMI.

8.3.1 CiGetAttribute

This function gets value(s) of an individual attribute or multiple attributes within a single group. Although the *DmiGetAttributes* command from the MI allows gets across multiple groups, the service provider must serialize calls across groups at the component interface level.

This function returns a pointer to a *DmiAttributeData_t* object that contains the ID, type, and pointer to value for the requested attribute. The component ID, group ID, and attribute ID are passed in as parameters.

If the given group is not a table, then *keyList* will be a NULL pointer. If the group is a table a *keyList* may or may not be given. If it is provided, then the attribute value from the requested row should be returned. If there is no key list, then the attribute value from the first row should be returned.

PARAMETER NAME	DIRECTION	DESCRIPTION
componendId	In	Component ID containing group
groupId	In	Group ID containing attribute
attributeId	In	Attribute ID to get
language	In	language-code | territory-code | encoding for return data
keylist	In	List of row keys
data	Out	Attribute value returned

```
DmiErrorStatus_t DMI_API
CiGetAttribute (
    [in]    DmiId_t                 componentId,
    [in]    DmiId_t                 groupId,
    [in]    DmiId_t                 attributeId,
    [in]    DmiString_t*            language,
    [in]    DmiAttributeValues_t*   keyList,
    [out]   DmiAttributeData_t**    data);
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_ATTRIBUTE_NOT_FOUND
DMIERR_COMPONENT_NOT_FOUND
DMIERR_GROUP_NOT_FOUND
DMIERR_ILLEGAL_KEYS
DMIERR_ILLEGAL_TO_GET
DMIERR_ROW_NOT_FOUND
DMIERR_ATTRIBUTE_NOT_SUPPORTED
DMIERR_VALUE_UNKNOWN
```

8.3.2 CiGetNextAttribute

This function gets the value of the attribute immediately proceeding the currently referenced attribute, returning a pointer to a DmiAttributeData_t object that contains the ID, type, and pointer to value for the SUCCESSOR of the specified attribute.

PARAMETER NAME	DIRECTION	DESCRIPTION
componendId	In	Component ID containing group
groupId	In	Group ID containing attribute
attributeId	In	Attribute ID to get
language	In	language-code | territory-code | encoding for return data
keylist	In	List of row keys
data	Out	Attribute value returned

```
DmiErrorStatus_t DMI_API
CiGetNextAttribute (
    [in]     DmiId_t             componentId,
    [in]     DmiId_t             groupId,
    [in]     DmiId_t             attributeId,
    [in]     DmiString_t*        language,
    [in]     DmiAttributeValues_t*       keyList,
    [out]    DmiAttributeData_t**        data);
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_ATTRIBUTE_NOT_FOUND
DMIERR_COMPONENT_NOT_FOUND
DMIERR_GROUP_NOT_FOUND
DMIERR_ILLEGAL_KEYS
DMIERR_ILLEGAL_TO_GET
DMIERR_ROW_NOT_FOUND
DMIERR_ATTRIBUTE_NOT_SUPPORTED
DMIERR_VALUE_UNKNOWN
```

8.3.3 CiSetAttribute

This function is called to set the specified attribute with the given value. The component ID, group ID, and attribute ID are passed in as parameters.

If the given group is not a table, then keyList will be a NULL pointer. If the group is a table a keyList may or may not be given. If it is provided, then the attribute in the specified row should be set. If there is no key list, then the attribute in the first row should be set.

PARAMETER NAME	DIRECTION	DESCRIPTION
componendId	In	Component ID containing group
groupId	In	Group ID containing attribute
attributeId	In	Attribute ID to get
language	In	language-code | territory-code | encoding for return data
keylist	In	List of row keys
data	In	Attribute value to set

```
DmiErrorStatus_t DMI_API
CiSetAttribute (
    [in]     DmiId_t             componentId,
    [in]     DmiId_t             groupId,
    [in]     DmiId_t             attributeId,
    [in]     DmiString_t*        language,
    [in]     DmiAttributeValues_t*       keyList,
    [in]     DmiAttributeData_t*         data);
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_OUT_OF_MEMORY
DMIERR_SP_INACTIVE
DMIERR_ATTRIBUTE_NOT_FOUND
DMIERR_VALUE_EXCEEDS_MAXSIZE
DMIERR_COMPONENT_NOT_FOUND
DMIERR_GROUP_NOT_FOUND
DMIERR_ILLEGAL_KEYS
DMIERR_ILLEGAL_TO_SET
DMIERR_ROW_NOT_FOUND
DMIERR_ATTRIBUTE_NOT_SUPPORTED
```

8.3.4 CiReserveAttribute

This function is called to query if the specified attribute could be set given that these same parameters were passed to the CiSetAttribute procedure. The function returns CiTrue or CiFalse.

PARAMETER NAME	DIRECTION	DESCRIPTION
componentId	In	Component ID containing group
groupId	In	Group ID containing attribute
attributeId	In	Attribute ID to get
keylist	In	List of row keys
data	In	Attribute value to reserve

```
DmiErrorStatus_t DMI_API
CiReserveAttribute (
    [in]     DmiId_t             componentId,
    [in]     DmiId_t             groupId,
    [in]     DmiId_t             attributeId,
    [in]     DmiAttributeValues_t*       keyList,
    [in]     DmiAttributeData_t*         data);
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_OUT_OF_MEMORY
DMIERR_SP_INACTIVE
DMIERR_ATTRIBUTE_NOT_FOUND
DMIERR_VALUE_EXCEEDS_MAXSIZE
DMIERR_COMPONENT_NOT_FOUND
DMIERR_GROUP_NOT_FOUND
DMIERR_ILLEGAL_KEYS
DMIERR_ILLEGAL_TO_SET
DMIERR_ROW_NOT_FOUND
DMIERR_ATTRIBUTE_NOT_SUPPORTED
```

8.3.5 CiReleaseAttribute

This function is called to request that the instrumentation code decommit from a set operation after a reserve has been issued.

PARAMETER NAME	DIRECTION	DESCRIPTION
componentId	In	Component ID containing group
groupId	In	Group ID containing attribute
attributeId	In	Attribute ID to get
keylist	In	List of row keys
data	In	Attribute value to release

```
DmiErrorStatus_t DMI_API
CiReleaseAttribute (
    [in]     DmiId_t             componentId,
    [in]     DmiId_t             groupId,
    [in]     DmiId_t             attributeId,
    [in]     DmiAttributeValues_t*       keyList,
    [in]     DmiAttributeData_t*         data);
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_OUT_OF_MEMORY
DMIERR_SP_INACTIVE
DMIERR_ATTRIBUTE_NOT_FOUND
DMIERR_VALUE_EXCEEDS_MAXSIZE
DMIERR_COMPONENT_NOT_FOUND
DMIERR_GROUP_NOT_FOUND
DMIERR_ILLEGAL_KEYS
DMIERR_ILLEGAL_TO_SET
DMIERR_ROW_NOT_FOUND
DMIERR_ATTRIBUTE_NOT_SUPPORTED
```

8.3.6 CiAddRow

This function allows component instrumentation to directly add a row of data to an existing table. This is simplified over the DMIv1.x model, which required instrumentation code to register with the MI for similar operations.

PARAMETER NAME	DIRECTION	DESCRIPTION
rowData	In	Attribute values to set

```
DmiErrorStatus_t DMI_API
CiAddRow (
    [in]     DmiRowData_t*rowData );
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_OUT_OF_MEMORY
DMIERR_SP_INACTIVE
DMIERR_VALUE_EXCEEDS_MAXSIZE
DMIERR_GROUP_NOT_FOUND
DMIERR_ILLEGAL_KEYS
DMIERR_UNABLE_TO_ADD_ROW
```

8.3.7 CiDeleteRow

This function allows component instrumentation to directly delete a row of data from an existing table.

PARAMETER NAME	DIRECTION	DESCRIPTION
rowData	In	Row data to delete (component, group, attribute)

```
DmiErrorStatus_t DMI_API
CiDeleteRow (
    [in]     DmiRowData_t*       rowData );
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_OUT_OF_MEMORY
DMIERR_SP_INACTIVE
DMIERR_ENUM_ERROR
DMIERR_GROUP_NOT_FOUND
DMIERR_ILLEGAL_KEYS
DMIERR_ROW_NOT_FOUND
DMIERR_UNABLE_TO_DELETE_ROW
```

9. OPTIONAL MI SUPPORT FUNCTIONS

The extensions presented here are optional and therefore not required for implementation.

DMIv2.0, a procedural interface to DMI, is remoteable via the use of RPCs. A DMI Client (Management Application) may need to communicate with multiple DMI Service Providers, not all of which support the same RPC. For example, a Windows NT machine would be reachable through DCE/RPC, while a UNIX machine might be reachable via SUN's ONC/RPC.

While clients can be written to support multiple RPCs, this is cumbersome and requires the client writer to invest in coding for communication purposes, rather than for managing the remote node. The MI Support Functions interface serves as a front end to hide RPC specifics from the client, thus enabling the client to concentrate on the managing aspect of the application. An explicit goal is to make client code written to the MI Support Functions easily usable under a specific RPC environment, requiring only slight modifications.

To achieve this, the MI Support Functions must address and hide RPC specifics such as:

- Connection establishment and tear-down
- Present a unified error model to the client, hiding RPC specific details
- Provide an API through which the client can issue DMI calls.
- Handle memory allocation and release to ease this burden for the user of the RPC mechanisms and to reduce the chance of introducing memory leaks.

This chapter presents the MI Support Functions, provided on the client side. It discusses a unified error model, both simple and extended, presents connection establishment and teardown helper functions, and applies them to run-time binding of RPC specific implementation of DMI.

9.1 PROGRAMMING CONSIDERATIONS

The intention in providing this abstraction layer is to isolate the user of the DMI from the intricacies of working with an RPC, and to allow the use of multiple RPCs. With that abstraction come a few programming considerations that must be kept in mind.

All memory used by the DMI Functions, and the application using those functions must be allocated and freed from a consistent heap. To accomplish this, the API provides a set of functions to allow just such memory management:

- DmiAllocPool()
- DmiFreePool()
- DmiAlloc()
- DmiFree()

The function of each of these APIs will be discussed in detail but, for now, it is important to keep in mind that when using the MI Support Functions APIs as an access method to the DMI, these memory management functions must be used to allocate and de-allocate memory used with this interface.

The use of memory is also a concern when dealing with incoming indications. To simplify this issue, a user of this interface should only consider a block of memory, passed on an indication, to be good for the duration of the call. During the indication call, the application should either copy the data, or complete all of the processing it plans to do with that data before returning from the call.

9.2 RPC ABSTRACTIONS

The MI Support Functions serve as a front end which provides all DMI functionality through multiple RPCs. To that effect, the MI Support Functions use the RPC specific DMI definition in order to communicate with the DMI Service Provider using that RPC. At the same time, the MI Support Functions present the client with the DMI API, as defined elsewhere in this document.

The MI Support Functions (a) present the DMI functional entry points as defined elsewhere in this document, to client application, as well as (b) use the DMI API to communicate with all RPC specific libraries. The following modifications are applied to the DMI API by the MI Support Functions:

- The error status is unified, to represent all error sources (DMI Service Provider, as well as RPC packages).
- Additional helper functions are provided to handle errors.
- Additional functions are provided for connection establishment and teardown.
- Additional memory management functions are provided to handle bulk allocation and de-allocation of memory across the interface.

In addition, RPC and platform specific client linkage is defined to enable run-time addition of RPC specific DMI implementations.

9.2.1 MI Support Functions and RPC specific DMI API

This chapter defines the API provided by the MI Support Functions. The DCE/RPC specific API, and the ONC/RPC specific API, which are used by the MI Support Functions, are described in their respective interface description languages and are attached as Appendices to this document.

9.3 CONNECTION ESTABLISHMENT AND TEARDOWN

The following functions are provided in order to facilitate connection establishment and teardown in a RPC independent fashion:

9.3.1 Connection Establishment

RPC Specific details of connection establishment are handled using this call. The result of this call is a Binding Handle. In addition to an error information storage area, the Binding Handle contains information about the Management Handle generated at the RPC stub interface when the MI Support Functions interface invokes remote DMI functions on behalf of the Management Application. This Management Handle is used in DmiRegister and subsequent DMI commands.

The *DmiIndicationFuncs* structure contains the address of indication callback functions provided by the Management Application. Incoming indications are handed to the Management Application at these entry points. There is one entry for each DMI indication type. The function prototypes are discussed in Section 7. If the application is not interested in a particular indication type, then it can pass a NULL value for that function's address to the MI Support Functions interface.

```
typedef  struct  DmiIndicationFuncs {
     DmiDeliverEvent*              DeliverEventFunc;
     DmiComponentAdded*            componentAddedFunc;
     DmiComponentDeleted*  componentDeletedFunc;
     DmiLanguageAdded*     languageAddedFunc;
     DmiLanguageDeleted*           languageDeletedFunc;
     DmiGroupAdded*                groupAddedFunc;
     DmiGroupDeleted*              groupDeletedFunc;
     DmiSubscriptionNotice*        subscriptionNoticeFunc;
} DmiIndicationFuncs_t;
```

Management Applications use the DmiBind function to bind themselves to the MI Support Functions interface and specify which particular machine they wish to correspond with and what transport and RPC to use on the connection. In return, they receive a Binding Handle of type `bind_handle_t`.

```
DmiErrorStatus_t DMI_API DmiBind (
     [out] bind_handle_t*            iMgmtHandle,
     [in]   char *                   rpc,
     [in]   char *                   transport,
     [in]   char *                   machine,
     [in]   DmiIndicationFuncs_t*    funcs
 );
```

Where *rpc* is the name of the RPC, and the *transport* is the name of the transport to use under that RPC. *rpc* and *transport* parameters are further defined in Section 9.3.3.[9] The Management Applications use their Binding Handles when invoking DMI functions through the MI Support Functions interface.

9.3.2 Connection Teardown

This call is used to close and release any resources allocated during connection establishment process.

```
DmiErrorStatus_t DMI_API DmiUnbind(
     [in] bind_handle_t iMgmtHandle
);
```

9.3.3 Transport List

The *transport* parameter in the Connection Establishment (Section 9.3.1), Connection Teardown (Section 9.3.2) and Indication Subscription (Section 9.3.1) entry points is an opaque string parameter that is passed through to the underlying RPC implementation to select the transport of interest.

[9] Note that the rpc name and transport name are also used to derive the name of the dynamically linked RPC specific library. See Section 9.5, Runtime Linkage, for more details.

Shown below is a list of some possible values for this parameter in the RPCs of interest. Note that not all possible values of the opaque string may be represented in the list below. There may be more recent additions to the list in the various standard RPCs, as well as in extensions to the standard RPCs by various RPC vendors.

RPC DESCRIPTION	TRANSPORT DESCRIPTION	FUNCTION
local	dmi	Local RPC used
dce OSF DCE/RPC	ncacn_ip_tcp	Connection-oriented TCP/IP
	ncadg_ip_udp	Datagram-oriented UDP/IP
onc SUN RPC	udp	UDP/IP
	tcp	TCP/IP
ti TI RPC (determined by /etc/netconfig, or equivalent file)	ticlts	Connectionless Loopback Transport Provider Interface
	ticots	Connection Oriented Loopback TPI (Transport Provider Interface)
	ticotsord	Connection Oriented Loopback TPI with orderly release
	tcp	Connection Oriented TCP/IP TPI with orderly release
	udp	Connectionless UDP/IP TPI
	rawip	Raw IP Protocol
	icmp	Internet Control Message Protocol

9.4 ERROR MODEL

To hide the RPC specifics details related to error handling, the MI Support Functions coalesce all error information into a single error return value. The MI Support Functions also provide extended error information, for clients interested in this information.

The DMI only provides error information in the form of error status returned. No support is provided for DCE/RPC exception mechanisms, or any other exception mechanisms.

9.4.1 Simple Error Handling

Simple error handling is targeted toward applications that are interested in the following information:

- Success/Fail status (including time-outs)
- Action Recommendation
- Error status
- Error text

Information is supplied using a set of C functions.

The model operates as follows. The management application calls a DMI procedure within the Optional MI Support Functions interface to accomplish a specific DMI function, e.g. GET the value of an attribute, SET the value of an attribute, etc. Upon returning, the procedure provides a return value to the management application of the type

```
error_status_t
```

This type is a composite structure[10] that conceptually contains three items, namely: a simple error result code, the full DMI error code as provided by the (potentially remote) DMI Service Provider, and the RPC error code that was returned by the underlying RPC implementation. The simple error result is characterized by the following enumeration definition and typedef:

```
enum error_result {
    DMI_RESULT_SUCCESS,
    DMI_RESULT_FAIL,
    DMI_RESULT_UNKNOWN,
};

typedef enum error_result error_result_t;
```

9.4.1.1 SUCCESS/FAIL STATUS

Whether or not the Management Application's call to the DMI functions succeeded or failed is ascertained by testing the return value against DMI_NO_ERROR.

For example:

```
status = DmiListComponents(...);
if (status != DMI_NO_ERROR ) {
    /* analyze/fail */
}
/* success */
```

[10] **NOTE** that the realization of error_status_t type is likely not to be made visible by the vendor of the MI Support Functions interface. The actual realization may vary between different implementations of the MI Support Functions. Code writers should only access error_status_t information using the provided functions.

9.4.1.2 ERROR STATUS - DmiErrorStatus

When the calling Management Application obtains a return value of type `error_status_t`, it submits this return value as an **in** parameter to an error interpretation function DmiErrorStatus that returns the error status.

DmiErrorStatus is defined as follows:

```
error_result_t DMI_API DmiErrorStatus(
    [in] error_status_t* status
);
```

The Management Application then compares the return from this function to DMI_RESULT_SUCCESS, DMI_RESULT_FAIL, or DMI_RESULT_UNKNOWN, to determine the nature of the result from the DMI procedure. If the result was DMI_RESULT_SUCCESS, then the application proceeds to its next operation. If, however, it encounters the codes DMI_RESULT_FAIL, or DMI_RESULT_UNKNOWN, it may take further action as follows.

9.4.1.3 ACTION RECOMMENDATION - DmiErrorAction

The Management Application next invokes the helper function DmiErrorAction with the structure of type `error_status_t` as an **in** parameter. In response, the DmiErrorAction function analyzes the RPC and DMI error codes contained within this **in** parameter and then returns an item of type `error_action_t` that is defined as follows:

```
enum error_action {
    DMI_ACTION_NORETRY,    /* do not retry      */
    DMI_ACTION_RETRY,      /* retry the command */
    DMI_ACTION_UNKNOWN,    /* need more info    */
    DMI_ACTION_NONE,       /* no action required */
};

typedef enum error_action error_action_t;
```

The DmiErrorAction () function is defined as follows:

```
error_action_t DMI_API DmiErrorAction(
    [in] error_status_t* status
);
```

The recommendation returned by DmiErrorAction might be any of the following:

- Do not retry the command. (DMI_ACTION_NORETRY)
- Re-try the command. (DMI_ACTION_RETRY)
- Unknown. (DMI_ACTION_UNKNOWN)
- No action required (DMI_ACTION_NONE)

9.4.1.3.1 DMI_ACTION_NORETRY - Do not retry

The command was sent to the remote node, and either failed at the remote node (Service Layer Error), or a communication error occurred while returning the information (The reason for this recommendation in this case is that the operation may yield undesirable results when an instrumentation code is re-executed.)

9.4.1.3.2 DMI_ACTION_RETRY - Re-try the command

The command was not sent, was not completely received, or there existed a condition at the remote Service Layer which prevented its execution. It is safe to re-try the command.

9.4.1.3.3 DMI_ACTION_UNKNOWN - Unknown

There was not sufficient information to determine in the command was received at the other end. The command may have been executed at the remote end, so decision taken must be based on extra error information or is related to the operation performed.

9.4.1.3.4 Error Action Example

As an example, this is how the Management Application might invoke DmiErrorAction:

```
do {
    status = DmiListComponents( ... );

    /* Handle remote DMI SL Errors here    */
    /* Need to break out if not comm error */

    if (comm_error) {
        break;
    }

    action = DmiErrorAction(status);

} while ( action == DMI_ACTION_RETRY );

if ( status != DMI_NO_ERROR ) {
    /* analyze/report error */
}
```

The combinations of success/fail status and action recommendations are summarized in the following table:

	STATUS = SUCCESS	STATUS = FAIL	STATUS = UNKNOWN
action = NO_RETRY	Command was successful. No need to reissue. (DMI_ACTION_NONE)	A communication error has occurred after command was completely sent or while receiving confirmation. Command executed at remote node. Recommendation is not to reissue the command, unless re-execution is permissible.	A communication error has occurred after command was successfully sent to the remote node. The command is known to have been received, but its execution status is unknown, however, it is assumed that if the command was valid, it was executed. Recommendation is not to re-issue the command.
action = RETRY	Command failed due to parameter error or execution error. All communications aspect of the command execution have been successful. Recommendation is to reissue with fixed parameters. (This is a DMI Service Provider error)	A communication error has occurred before command was completely sent. Command not executed at remote node. Recommendation is to reissue the command.	A error has occurred while command was sent to the remote node. However, It is known that the command has not been fully received, thus it was not executed at the remote end. Recommendation is to reissue the command.

	STATUS = SUCCESS	STATUS = FAIL	STATUS = UNKNOWN
action = UNKNOWN	N/A	N/A	A communication error has occurred while command was sent to the remote node. It is unknown if the command was received and executed. Recommendation is to further investigate, based on extended error information.

9.4.1.4 ERROR CODES - DMIERRORCODE AND DMIRPCERRORCODE

The main error status (in case of an error), whether it is a DMI Service Provider error code, or an underlying RPC error code, is returned using the DmiDmiErrorCode() and DmiRpcErrorCode() functions:

```
DmiUnsigned_t DMI_API DmiDmiErrorCode(
    [in] error_status_t* status
);

DmiUnsigned_t DMI_API DmiRpcErrorCode(
    [in] error_status_t* status
);
```

Error status returned include Service Provider errors, in addition to RPC specific error codes.

9.4.1.5 ERROR TEXT - DMIERRORTEXT

This function returns a static string which can be used to display/log errors. The string is localized as per the sLanguage set for the specific management handle used when the error occurred, or is an ISO 8859-1 string if the handle is not valid (as is the case before connection establishment or after connection has been terminated):

```
const char* DMI_API DmiErrorText(
    [in] bind_handle_t*    handle,
    [in] error_status_t*   status
);
```

9.4.2 Extended Error Handling

Applications interested in further information may access the unified error information structure. Information gathered is contained in a static array of structures, each containing error information as provided by the specific RPC, together with whatever other relevant information available. Access to the structure is available using DmiGetExtendedError() function.

The DmiGetExtendedError() may return NULL to indicate that no extended error information is available. Such implementation should not be regarded as non-compliant.

This function returns an item of type DmiExtendedError which is, in effect, a pointer to a per-session extended error status structure. Shown below is a possible example of such an extended error structure. **NOTE: this is simply an example and applications must not depend on the structures necessarily having this form. Applications must use functions provided by the MI Support Functions Interface to access information within this structure.**

```
struct DmiExtendedError {
     struct DmiExtendedError *next;
     void *additional_information;
     void (*error_function)(
              int operation,
              struct DmiExtendedError *error,
              void *additional_information);
```

```
    unsigned long action;
    struct {
             int length;
             char *data;
    } remote_machine;
    char *remote_machine_name;
    char *subsystem_name;
    char *subsystem_description;
};

typedef struct DmiExtendedError DmiExtendedError_t;

DmiExtendedError_t * DMI_API DmiGetExtendedError(
    [in] bind_handle_t;
);
```

9.4.2.1 NEXT

A pointer to the next member of the extended error information list. a NULL pointer signals the end of the list. Returned by the function:

```
DmiExtendedError_t * DMI_API DmiNextExtendedError(
    DmiExtendedError_t * extended_error;
);
```

9.4.2.2 ADDITIONAL INFORMATION AND ERROR_FUNCTION

This is a pointer to additional information about the error, which can only be interpreted by subsystem specific routine. Each subsystem which makes use of such information should also provide an error handling function, `error_function`, which takes this information as one of its inputs. The implementation of this function and linkage to it will be operating system specific.

This `error_function` implements the subsystem specific error handling which is targeted in re-establishing proper working order of the subsystem. The input to this function is the operation required, a pointer to the current error information structure and the subsystem additional information data. This function may modify the global error information structure, remove or add elements to it, as required. Further definition of the parameters is subsystem specific.

A typical example of a subsystem might be a specific RPC and transport combination used.

9.4.2.3 ACTION

This is an enumeration, specifying the recommended action that a management application should take. This information is derived from other sources, as appropriate for the transport and RPC used. Returned by:

```
error_action_t * DMI_API DmiExtendedErrorAction (
    DmiExtendedError_t extended_error
);
```

9.4.2.4 REMOTE MACHINE

This is a designation of the remote machine where the error occurred, in a machine usable manner (i.e., the information can be used to access the remote machine where the error occurred.)

9.4.2.5 REMOTE MACHINE NAME

This is a printable representation of the above, for error reporting purposes.

9.4.2.6 SUBSYSTEM_NAME

This is the subsystem name where the error occurred, for reporting purposes.

9.4.2.7 SUBSYSTEM_DESCRIPTION

This is the subsystem description, for reporting purposes.

9.4.3 DCE/RPC and ONC/RPC mapping for standard functions

OP	DMI	ONC/TI RPC	DCE RPC
success/fail test	`!= DMI_NO_ERROR`	`!= 0`	`!= rpc_s_ok`
Action	DmiErrorAction()	-	-
Error number	DmiErrorStatus()	`re_status` member of `rpc_err`.	DmiErrorStatus_t returned upon call.
Error Text	DmiErrorText()	clnt_sperrno()	dce_error_inq_text()
Extended error info.	DmiGetExtendedError()		

Extended Error information:

EXTENDED ERROR MEMBER	ONC/TI RPC	DCE RPC
error	`re_status`	(returned at call)
error_string	clnt_sperrno()	dce_error_inq_text()
additional_information	`rpc_err`	
action	(generated)	(generated)
remote_machine	(generated)	(generated)
remote_machine_name	(generated)	(generated)
subsystem_name	(generated)	(generated)
subsystem_description	(generated)	(generated)

9.5 RUNTIME LINKAGE

The MI Support Functions implementation may either statically support a pre-defined list of RPCs, or may apply run-time linkage to gain access to other RPC code. RPC binding is accomplished using the DmiBind() call, as follows:

```
DmiErrorStatus_t DMI_API DmiBind(
    NULL, rpc, transport, NULL );
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
```

Where *rpc* is the rpc name, used to derive the DLL/share object containing the RPC specific DMI code and *transport* is the transport. A statically linked implementation should return 0 if the transport exists, or should otherwise signal an error condition.

RPC transports are unbound implicitly as a result of a call to the DmiUnbind() function, as follows:

```
DmiErrorStatus_t DMI_API DmiUnbind(
    DmiUnsigned_t handle);
```

Where *handle* is assigned at bind time.

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_ILLEGAL_HANDLE
DMIERR_OUT_OF_MEMORY
DMIERR_INSUFFICIENT_PRIVILEGES
DMIERR_SP_INACTIVE
```

9.5.1 Naming Conventions

The name of the RPC specific DMI client library is as follows:

PLATFORM	LIBRARY NAME
UNIX	dmirpc.so
Netware	dmirpc.nlm (rpc name 4 chars max)
Win16	dmirpc16.dll (rpc name 3 chars max)
Win32	dmirpc32.dll (rpc name 3 chars max)
OS/2	dmirpc.dll

Where *rpc* stands for one of:

RPC	STANDARD NAME
DCE/RPC	dce
ONC/RPC	onc
TI/RPC	ti
LOCAL	local

Since some OS allow only a single name space for all shared libraries, some OS specific libraries will require that all DMI function names be prefixed with the RPC name. The following tables indicates where such prefix is required. In all other cases, the exported function names should match EXACTLY the functions defined in the Procedural MI section of this document.

PLATFORM	PREFIX NAMES
UNIX	not required
Netware	required
Win16	not required
Win32	not required
OS/2	not required

9.5.2 Runtime linkage example

One interesting example of how runtime linkage may be used to extend DMI to use other RPC is the case of a local, no-rpc implementation. A local implementation needs to provide a dynamically linked library, properly named as per the operating system used (for example, Win16 implementation would use DMILOC16.DLL .) This library, presenting a DMI compatible interface, would be linked under the MI Support Functions, and would thus be accessible to any Management Application/Client.

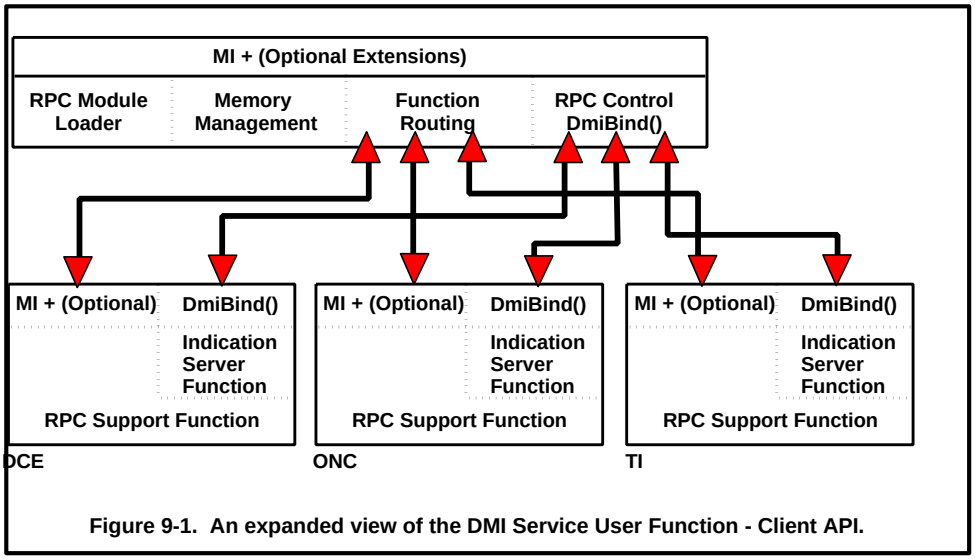


Figure 9-1. An expanded view of the DMI Service User Function - Client API.

The user (Management Application) in all cases will see only the MI interface exposed by the DMI Service User Function, for sake of clarity let's call it a DLL. This DLL is responsible for loading and managing all of the RPC functions (again let's think of them as DLLs) below it. Not only is the User function DLL responsible for loading the RPC DLLs when needed, but it is also responsible for managing the function routing tables that will be required to pass the calls through to the correct RPC DLL.

The DmiBind() function carries information in it that must be passed to the RPC DLL - namely the indication entry point information. The DMI Service User function (DLL) is also an RPC Server, in that it has to field indications. It must have a way of forwarding those received indications up to the application. This is where the DmiBind() call plays a role. This call carries the entry point information for indications in it. See the description of that function in Section 9.5.

9.6 MEMORY HANDLING FUNCTIONS

The MI Support Functions provide the client writer with convenient memory allocation routines, in order to ease memory handling and allocation. DMI associates allocated memory to *pools*, being a convenient way of grouping allocated memory. Users may create pools, allocate memory and associate it to a specific pool or free pool memory. Pools can also be destroyed; this would also cause all allocated memory belonging to that pool to be released.

9.6.1 DmiAllocPool

This function is used to create a pool of memory. Subsequent calls to DmiAlloc() should use a memory pool handle to associate allocated memory with that pool:

```
DmiVoid_t* DMI_API DmiAllocPool(
    void
);
```

The function return value is a pool handle, to be used in subsequent DmiAlloc() calls. DmiAllocPool() should return NULL is memory pool cannot be created.

Note that multiple active pools can exists at the same time.

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_NO_POOL
```

9.6.2 DmiAlloc

This function is used to allocate memory for use as input parameters to DMI calls, or any other transient use. It prototype is:

```
DmiVoid_t* DMI_API DmiAlloc(
    [in] DmiVoid_t *       pool_handle,
    [in] DmiUnsigned_t     size
);
```

Where pool_handle is the handle returned by DmiAllocPool(), and size is the number of bytes to allocate.

The DmiAlloc() function should return NULL if memory cannot be allocated.

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_INVALID_POOL
DMIERR_OUT_OF_MEMORY
```

9.6.3 DmiFree

This function is used to free previously allocated memory:

```
DmiErrorStatus_t DMI_API DmiFree(
[in] DmiVoid_t *           ptr
);
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_INVALID_POOL
DMIERR_INVALID_PTR
```

9.6.4 DmiFreePool

Memory allocated using DmiAlloc() which belongs to a specific pool can be released using DmiFreePool() call. This call would also delete the specified pool:

```
DmiErrorStatus_t DmiFreePool(
[in] DmiVoid_t *           handle
);
```

ERROR CODES

```
DMIERR_NO_ERROR
DMIERR_INVALID_POOL
```

9.6.5 Bulk Allocation

DmiAllocPool, DmiAlloc and DmiFreePool can be used to ease memory allocation tracking. A DMI Client may use DmiAllocPool() to create a memory pool, and request that memory allocated using the DmiAlloc() function be owned by it. Memory belonging to that pool can then be freed using DmiFreePool(). For example:

```
manage_client(){
     DmiVoid_t *h, *h1,*h2;
     DmiErrorStatus_t status;

     h = DmiAllocPool();
     ...
     h1 = DmiAlloc(h, 100UL); /* allocate h1 */
     ...
     h2 = DmiAlloc(h, 200UL); /* allocate h2 */
     ...
     status = DmiListComponents(...)
     ...
     DmiFreePool( h ); /* free h1, h2, h */
}
```

Using DmiFreePool releases the client writer from tracking all allocated memory, and provides an easy way of preventing memory leakage problems common to RPC code.

10. INTRODUCTION TO DMI2.0S

DMIv2.0s defines a mechanism to control remote access to the ***DMI*** Management Interface and local access to ***DMI*** interfaces. The remote access control mechanism is defined on top of standard RPC mechanisms, whereas the local access control mechanism is defined on top of operating system mechanisms. ***DMIv2.0s*** does *not* specify a standard format for identities nor a cryptosystem to verify those identities, but relies on those provided through the RPC and by the operating system. In addition, ***DMIv2.0s*** defines that certain operations performed by the DMIv2.0s Service Provider may be logged and/or generate indications. The DMI Security Extension introduced by ***DMIv2.0s*** appear in Sections 10 through 18.

DMIv2.0s Service Providers should be compatible with existing ***DMI*** management applications and component instrumentation. The functions and parameters of the Management Interface and the Component Interface in ***DMIv2.0s*** are identical to those of ***DMIv2.0***; that is, the IDL of ***DMIv2.0s*** is identical to that of ***DMIv2.0***. ***DMIv2.0s*** adds authentication features to the remote Management Interface invocation mechanism, and specifies that the DMIv2.0s Service Provider authorizes commands according to the identity of the user accessing the Management Interface. Access to the Component Interface and to the local Management Interface can be restricted to privileged users. The DMIv2.0s Service Provider can be configured to log and generate indications upon certain security-related operations. ***DMIv2.0s*** also defines the behavior of a DMIv2.0s Service Provider in the presence of non-authenticated management applications.

10.1 OVERVIEW

The ***DMI*** architecture defines the Service Provider, a program that runs on the managed system, and communicates with management applications by means of the Management Interface and with managed components by means of the Component Interface. ***DMIv2.0*** uses a standard Remote Procedure Call mechanism to expose the Management Interface to remote management applications. Because ***DMIv2.0*** does not define security mechanisms to control access to the various elements of the ***DMI***, an unauthorized user could invoke a standard ***DMIv2.0*** management application from any computer on the network. With the growing number of ***DMI***-enabled systems deployed in the market, there is a strong demand by vendors and users for a more secure version of ***DMI***. In response to this request, the DMTF has formed the DMI Security Working Committee which is chartered with extending the ***DMIv2.0*** specification for security.

DMIv2.0s is a standard extended version of the ***DMIv2.0*** specification. ***DMIv2.0s*** defines mechanisms to secure the interaction between the Service Provider, management applications, component instrumentation and the Management Information Format (MIF) database. In order to describe the features of ***DMIv2.0s***, we will use several terms related to security in a networked computing environment such as *authentication* and *authorization*. Refer to Appendix E for a definition of those and other terms.

DMIv2.0s defines the following features to control and track the interactions between ***DMI*** elements:

- control access of remote management applications to ***DMI*** information
- security of component instrumentation
- security of MIF database
- security of local management applications
- generating events upon security-related operations
- logging of security-related operations
- role-based authorization model
- flexible, remotely configurable authorization policy
- implementing of the authentication interface on top of operating system or third party product

The approach followed to define these features is presented in Section 10.2.

Section 11 Architecture describes the ***DMIv2.0s*** extensions to the ***DMIv2.0*** specification: the functional blocks of ***DMIv2.0s***, the interfaces defined by ***DMIv2.0s***, the ***DMIv2.0s*** standard groups in the Service Provider component, and the standard roles defined by ***DMIv2.0s***.

Section 12 DMiv2.0s Service Provider standard groups describes several standard groups that must be included in the Service Provider component, such as the `SP Indication Subscription` group and the `SP Filter Information` group and introduces new standard groups to configure new features of the DMIv2.0s Service Provider and to store the authorization policy.

Section 13 Management interface security defines this main feature of ***DMIv2.0s***. Management Interface security controls the access of management applications to ***DMI*** data and instrumentation.

Section 14 Component interface security defines security as it applies to component instrumentation interfacing with the DMIv2.0s Service Provider, be it DMIv1 component instrumentation or DMIv2 component instrumentation.

Section 15 MIF Database PROTECTION defines the use of operating system or file system mechanisms to protect the MIF database from access by non-privileged users.

Section 16 Security Indications describes security indications to be sent to monitoring management applications.

Section 17 Logging describes security logging entries logged by the DMIv2.0s Service Provider for future retrieval by monitoring applications at their convenience.

The actual mechanisms used by the RPC infrastructure to authenticate users (e.g. passwords, X.509 digital certificates, SIDs, etc.) are outside the scope of this specification. This specification does not address threats from hackers that have access to hardware within a managed system (e.g. physical memory, virtual memory, buses, disks).

10.2 THE DMIv2.0S APPROACH

DMI defines a client-server model in which management applications are clients and the Service Provider is the server: management applications invoke ***DMI*** commands which are serviced by the Service Provider. Note that in the case of indication delivery the roles are reversed: the Service Provider initiates the delivery of indications to management applications which handle them.

In ***DMIv2.0s***, the Service Provider controls access to management information through the remote Management Interface according to a configurable policy. Management applications and component instrumentation have to authenticate with the Service Provider to be granted access. Each of these aspects is defined in the following paragraphs. A more technical description of ***DMIv2.0s*** features is found in Section 11.

10.2.1 Authentication

Authentication is a protocol through which a management application proves the identity of its user to the Service Provider, in order to be granted privileges according to the user's identity. ***DMIv2.0s*** does not specify an authentication method and name space. Instead, ***DMIv2.0s*** implementations can use any existing authentication method (often including user names, IDs, and passwords) available through an RPC infrastructure, thus saving the costly deployment and management of a new authentication framework. An example of a widely-deployed authentication system is the operating system. In most environments, users are defined in the context of the operating system and are authenticated upon logging on their system. ***DMIv2.0s*** may be implemented on top of an operating system authentication mechanism, so that a management application authenticates with the DMIv2.0s Service Provider according to the identity of the user invoking the management application. ***DMIv2.0s*** may also be implemented on top of an authentication system independent of any operating system such as Kerberos or X.509 certificates.

NOTE that a ***DMIv2.0s*** management application has to use an authentication method supported by the DMIv2.0s Service Provider on the managed system. For example, to access a DMIv2.0s Service Provider that uses X.509 certificates for authentication, a management application has to invoke the ***DMI*** Management Interface through an RPC that performs authentication using X.509 certificates.

10.2.2 Roles

In midsize and large installations, various groups of system administrators are in charge of managing different aspects of a computing system. Each group of administrators needs to be assigned a specific set of privileges. On the other hand, administrators frequently move from one group to another and assume different responsibilities, so their privileges need to be updated. Using *roles*, ***DMIv2.0s*** allows granting the same privileges to several users according to their **function** in managing the system.

A *role* is a set of privileges associated to a group of users. A user is said to possess a list of *roles*. Authentication yields the list of a user's roles, which is then used by the DMIv2.0s Service Provider for authorization. Implementations of ***DMIv2.0s*** that are based on operating system authentication can use operating system user groups to associate users with roles.

In addition to assigning the same role to several users, the roles paradigm allows associating the same role and privileges to users from different environments. For example, authentication may associate the same role to the group of UNIX helpdesk users and to the group of NT helpdesk users. Similarly, authentication may associate the same role to NT administrators (members of the Administrators group) and to UNIX administrators (members of group 0).

10.2.3 Policy

The *policy* determines which commands can be performed on which objects by which roles. The DMIv2.0s Service Provider looks up the policy to determine whether a ***DMI*** command invoked by a remote management application should be performed or rejected. The policy is stored as a table in the MIF database, and it can be accessed and protected as a regular ***DMI*** table. Each row in the table represents a policy statement which grants or denies the privilege of a role to perform a ***DMI*** command.

The policy enables the system administrator to “secure” an attribute by specifying the roles that can access it. If the policy “secures” an attribute, then only those roles specified will be granted access. Otherwise, if the policy does not “secure” the attribute, all roles will be granted access to that attribute. Since ***DMI*** defines standard groups (rather than standard attributes or standard components), attributes are identified in the policy by their group class string and their attribute ID. For example, it is possible to set a policy that allows only the helpdesk role to modify the base address of a serial port by defining a policy for attribute ID 2 in groups whose class string is `"DMTF|Serial Ports|003"`.

A policy row that specifies only an attribute ID and a group class string applies to all the groups in the system whose class string matches. To narrow down the policy row to apply only to a subset of those groups, an additional class, attribute ID, and value can be specified. In this case, the policy row will apply only to those components in which the value of the specified attribute matches the value in the policy. For example, it is possible to specify a different policy for each network interface card in a system, according to manufacturer or serial number.

The policy also enables the system administrator to specify which roles are allowed to perform database administration functions such as `DmiAddGroup` or `DmiDeleteComponent`.

10.2.4 Authorization

Authorization is the mechanism whereby the DMIv2.0s Service Provider decides whether to perform or reject a ***DMI*** command. The decision depends on the command, its parameters, the user’s roles, and the policy. Commands rejected return with status `DMIERR_INSUFFICIENT_PRIVILEGES`. Since a user may have several roles, a command is allowed if at least one of the user’s roles is allowed to perform the command. Thus, a user with several roles actually enjoys the combination of the privileges granted to each role.

To determine whether a role is authorized to perform a command, the DMIv2.0s Service Provider searches the policy table for rows that match the attempted command. If no such row is found, the command is allowed to all roles. Otherwise, the role is allowed to perform the command if there is (at least) one matching row that grants the role permission to perform the command and there is no matching row that denies the role permission to perform the command.

10.2.5 Logging and event generation

The DMIv2.0s Service Provider can be configured to log commands and to generate events upon several operations such as installation of components and registration of management applications. *Logging* and *event generation* are useful to detect security breaches in real time and to track actions that may affect the configuration of a system, and to keep users accountable for their actions. ***DMIv2.0s*** defines a logging interface which the Service Provider invokes when needed. The log format is defined by the logging module provided as part of the DMIv2.0s Service Provider. The rationale for not specifying the log format is that several such mechanisms exist and system administrators are familiar with them (e.g. syslog on UNIX, the event log on WinNT or AUDITCON on NetWare).

10.2.6 Security of local interfaces

DMIv2.0 defines that the Management Interface can be accessed through a Remote Procedure Call. The Management Interface can also be accessed locally (without going through an RPC) by directly invoking the appropriate entry point of the DMIv2 Service Provider. The ***DMIv1*** Management Interface and the ***DMIv1*** and ***DMIv2*** Component Interfaces are also local interfaces. Communication between the Service Provider and the MIF database, though not a programming interface, is also considered a local interface from the security point of view. Therefore, ***DMIv2.0s*** defines an elementary security model for local ***DMI*** interfaces: the MIF database, the local Management Interface and the local Component Interface are accessible only to privileged users.

Privileged users are defined by each operating system. Processes executed by privileged users are allowed to configure the operating system and the file system. The table below summarizes the definition of privileged users for several operating systems.

OS	PRIVILEGED USERS
UNIX	effective user ID is 0
NetWare	user is Supervisor or Admin
WinNT	user is member of NT administrators group
Win9x	all users are privileged

Thus, in ***DMIv2.0s***, privileged users are authorized to invoke any Management Interface command through the local Management Interface. (In the context of this specification, invoking the Management Interface through an RPC from the same system on which the Service Provider is running is *not* considered a local access, and the security model applied is the same as when the Management Interface is invoked through an RPC from a system different from the one running the Service Provider.)

10.2.7 OS dependence

DMI can be implemented on various operating systems, RPC flavors, and computer architectures. ***DMI*** specifications define interfaces and their behavior. These specifications do not define the specific mechanisms involved in implementing those interfaces and accessing them within a system (for example, calling convention, parameter passing, endianness). The local interfaces to access ***DMI*** under a specific architecture and operating system are defined by each Service Provider implementation; that is, calling conventions, parameter passing, and endianness are implementation-specific. Remote access is specified, though. Remote procedure calls to ***DMIv2.0*** Management Interface procedures are defined for each RPC flavor: the ONC and DCE RPC standards, along with the IDL and RPCGEN listings in the ***DMIv2.0*** specification define how to remotely access the Management Interface of ***DMIv2.0***.

DMIv2.0s requires that the Remote Procedure Calls be authenticated, but the specific authentication mechanism to use is determined by each DMIv2.0s Service Provider implementation. A ***DMIv2.0s*** management application has to use an authentication method supported by the DMIv2.0s Service Provider on the managed system. Authentication protocols may or may not be based on operating system mechanisms.

NOTE that even if the authentication mechanism supported by an implementation of the DMIv2.0s Service Provider is based on the operating system on which the Service Provider runs, management applications running under a different operating system may perform the authentication protocol. For example, just as a Windows user can log on to a NetWare server, a user running a management application on a Windows system can authenticate to a DMIv2.0s Service Provider running on a NW server using the NetWare login as authentication mechanism.

10.2.8 Compatibility

The Management Interface defined by the ***DMIv2.0*** is a remotable procedural interface (through a Remote Procedure Call mechanism), whereas the Component Interface is a local procedural interface. The actual mechanism used for local invocation of the Management Interface and the Component Interface is defined by each DMI Service Provider implementation. In ***DMIv1***, both the Component Interface and the Management Interface are local data block interfaces. The actual mechanism for invoking these data block interfaces is defined by each DMIv1 Service Provider implementation.

In ***DMIv2.0s***, the entry points and parameters of the Management Interface and the Component Interface are identical to those of ***DMIv2.0***. ***DMIv2.0s*** requires that the user invoking the Management Interface be authenticated through the RPC if access is remote or be a privileged user if access is local. ***DMIv2.0s*** requires that the user invoking the Component Interface be a privileged user. Authentication failures result in error codes.

The DMIv2.0s Service Provider authorizes commands according to the identity of the caller. If a command is authorized, its result is as defined in ***DMIv2.0***; if a command is not authorized, error code `DMIERR_INSUFFICIENT_PRIVILEGES` is returned and the command is not performed. Note that `DMIERR_INSUFFICIENT_PRIVILEGES` is defined by the ***DMIv2.0*** specification and, therefore, should be handled properly by existing management applications written to ***DMIv2.0***. Additionally, ***DMIv2.0s*** specifies that the Service Provider can be configured to log and generate indications upon certain operations. ***DMIv2.0s*** also defines the behavior of a DMIv2.0s Service Provider in the presence of component

instrumentation and management applications whose caller cannot be authenticated (management applications that do not use an authenticated RPC fall in this category).

Since one of the objectives of this specification is to allow a smooth transition to ***DMIv2.0s***, DMIv2.0s Service Providers will be compatible with existing ***DMI*** management applications and component instrumentation. For compatibility with existing component instrumentation and management applications, it is recommended that Service Provider writers offer implementations of ***DMIv2.0s*** that are binary compatible with their implementations of ***DMIv2.0***. It is recommended that DMIv2.0s Service Providers be able to read a MIF database generated by a DMIv2.0 Service Provider, so that ***DMIv2.0*** systems can be upgraded to ***DMIv2.0s*** without having to reinstall and configure each component.

11. ARCHITECTURE

This section describes the ***DMIv2.0s*** extensions to the ***DMIv2.0*** specification: the functional blocks of ***DMIv2.0s***, the interfaces defined by ***DMIv2.0s***, the ***DMIv2.0s*** standard groups in the Service Provider component, and the standard roles defined by ***DMIv2.0s***.

NOTE that the partition into functional blocks or modules is intended to clarify the functionality of the DMIv2.0s Service Provider and not to impose an architecture on DMIv2.0s Service Provider implementations.

DMIv2.0s implements all the interfaces defined by ***DMIv2.0***, and specifies one additional interface: the Logging Interface which the DMIv2.0s Service Provider invokes in order to log operations and exceptional conditions. The semantics of existing ***DMIv2.0*** interfaces are extended by ***DMIv2.0s***: for example, commands that would have been executed by a DMIv2.0 Service Provider will be rejected by a DMIv2.0s Service Provider if the user invoking the command does not have the required privilege. Existing ***DMIv2.0*** management applications are supported in ***DMIv2.0s***. Management applications using a non-authenticated RPC infrastructure will be allowed to perform commands that the policy allows role `dmi_default` to perform.

11.1 DMIv2.0S FUNCTIONAL BLOCKS

11.1.1 Authentication

Authentication is performed at the time of management application registration. When a remote management application registers with the DMIv2.0s Service Provider, the RPC infrastructure authenticates the user. If authentication fails, the RPC infrastructure returns an RPC specific error. If authentication succeeds, the authentication module of the DMIv2.0s Service Provider retrieves the identity from the RPC infrastructure and yields the list of roles of the user. The authentication module may extract the roles list from the identity or it may retrieve it from a database. The actual mechanism used to associate a role with a user is defined by the DMIv2.0s Service Provider implementation. We recommend using operating system user groups or digital certificate attributes to map user identities to roles since system administrators are likely to be familiar with user/certificate administration and related tools.

The DMIv2.0s Service Provider associates the list of roles with the ***DMI*** management handle; that is, the roles list assigned at registration applies to all subsequent commands issued with that management handle. Optionally, the DMIv2.0s Service Provider may also perform authentication on each of the subsequent Management Interface RPC calls after `DmiRegister`, and compare the identity of the caller with the identity of the caller of `DmiRegister`; if different the service provider returns error `DMIERR_ILLEGAL_HANDLE`. Management applications that register with the Service Provider using a non-authenticated RPC will be assigned a role list that contains only role `dmi_default`.

If, during a ***DMI*** management session, the credentials of a management application expire or are revoked, the RPC infrastructure will reject all subsequent remote procedure calls, even if the DMIv2.0s Service Provider does not perform authentication at every call.

11.1.2 Authorization

For each ***DMI*** command issued by a management application, the DMIv2.0s Service Provider checks whether that management application is allowed to perform the command according to the management application role, the current contents of the Service Provider policy table and the command parameters.

11.1.3 Indication generation and logging

The DMIv2.0s Service Provider can be configured to generate indications upon some operations performed by management applications. These indications can be used to warn a system administrator of an operation that may endanger a system or alter its configuration.

The logging module of the DMIv2.0s Service Provider implements the Logging Interface defined in Section 17.1. The DMIv2.0s Service Provider can be configured to invoke this interface in order to log operations performed by management applications in a log. The log can be used to keep users accountable for their actions or to keep track of changes in the configuration of a system.

11.1.4 MIF database security

Since the policy is stored in the MIF database, it is necessary to protect the database. The contents of the MIF database are persistent across reboots and, therefore, the MIF database must be kept in some type of persistent storage, typically a file. The contents of the database are protected from unauthorized access by ***DMI*** management applications through the ***DMIv2.0s*** policy itself. However, it is also necessary to protect the database in its stored form, such as a file. A DMIv2.0s Service Provider must protect the MIF database from access by non-privileged users through file system mechanisms if supported by the system. If the MIF database is not stored as a file, an appropriate access control mechanism should be set if supported.

11.1.5 Component instrumentation security

Since component instrumentation controls the actual behavior of ***DMI*** instrumented components, it is one of the most powerful and vulnerable elements in the system. The DMIv2.0s Service Provider controls access of management applications to component instrumentation through the authorization mechanism of the Management Interface. However, it is also required to protect the Service Provider from unauthorized component instrumentation. The DMIv2.0s Service Provider can be configured to disable registration of component instrumentations that are not privileged (since privileged instrumentation is trusted by the OS).

The DMIv2.0s Service Provider can also be configured to disable overriding of component instrumentation by a subsequent registration of instrumentation for the same attribute.

12. DMIv2.0S SERVICE PROVIDER STANDARD GROUPS

The DMI Service Provider is itself a component of a system and it has an associated MIF that describes its capabilities. This component has a component ID equal to 1 by definition. Several standard groups are defined that must be included in the Service Provider component, such as the `SP Indication Subscription` group and the `SP Filter Information` group. ***DMIv2.0s*** introduces new standard groups to configure new features of the DMIv2.0s Service Provider and to store the authorization policy. These groups are described in the following sections.

NOTE that in the following group listings:

The group ID is included for syntactic correctness and is not part of the definition; instead, the groups should be identified by their class string.

Value statements in the table definitions define the default value of attributes omitted in a table initialization and should not be changed.

Value statements in scalar groups are the recommended initial value of the attribute. DMIv2.0s Service Provider implementations may choose to use different initial values.

12.1 DMIv2.0S SERVICE PROVIDER CONFIGURATION

The features provided by the DMIv2.0s Service Provider can be enabled or disabled through the “Service Provider Characteristics” group. The DMIv2.0s Service Provider checks the value of these boolean attributes upon startup and enables or disables features accordingly. A concise description is provided with each attribute. Access to attributes in the `Service Provider Characteristics` group is controlled by the policy like any other attribute. It is recommended that only administrators be allowed to modify these attributes.

```
Start Group
  Name = "Service Provider Characteristics"
  Class = "DMTF|SP Characteristics|001"
  ID = 6
  Description = "This group configures the DMIv2.0s SP characteristics."
```

The first attribute "`enable local security`" controls whether the DMIv2.0s Service Provider secures the local interfaces. If the value of this attribute is `True` when the Service Provider initializes, local interfaces are secured, thus:

- Component instrumentation which is not privileged cannot access the DMIv2.0s Service Provider
- A local management application which is not privileged cannot access the DMIv2.0s Service Provider

```
Start Attribute
    Name = "enable local security"
    ID = 1
    Type = start enum
       0x00 = "False"
       0x01 = "True"
    end enum
    Storage = common
    Value = "True"
End Attribute
Description = "If true, CI and MA must be privileged processes to "
     "access the DMIv2.0s SP.\"
```

The second attribute "`disable CI override`" controls whether the DMIv2.0s Service Provider allows component instrumentation registration to override a previous component instrumentation registration of the same attribute. If the value of this attribute is `True` when the Service Provider initializes, attempts to override a previous component instrumentation registration will fail with error `DMIERR_INSUFFICIENT_PRIVILEGES`.

```
Start Attribute
    Name = "disable CI override"
    ID = 2
    Description = "If true CI override attempts will fail."
    Type = start enum
       0x00 = "False"
       0x01 = "True"
    end enum
    Storage = common
    Value = "True"
End Attribute
```

Changes in `enable local security` and `disable CI override` take effect at the next Service Provider restart.

12.2 DMIv2.0S SECURITY INDICATION AND LOGGING CONFIGURATION

Security indication and logging are controlled by the `Service Provider Logging and Security Indication Characteristics` group. The first attribute `commands` determines which commands/occurrences are to be processed (Note that all ***DMI*** listing commands are grouped together.) The second attribute `level` determines under what success/failure conditions the specified commands are to be processed. Commands returning `DMIERR_NO_ERROR` or `DMIERR_NO_ERROR_MORE_DATA` are considered successful; Commands returning `DMIERR_INSUFFICIENT_PRIVILEGES` or `DMIERR_INVALID_HANDLE` are considered security failures; Commands returning other values are considered to have failed for non-security reasons. The third attribute `action` determines the type of processing: logging, security indication or both. The fourth attribute `class string filter` provides the ability to filter for what groups the processing is done. The semantics of this filter are similar to the class string parameter to the `ListComponentsByClass` command in the Management Interface.

```
Start Group
  Name = "Service Provider Logging and Security Indication Characteristics"
         Class = "DMTF|SP Logging and Security Indication Characteristics|001"
  Key = 1,2,3,4
  Description = "This table selects which commands are logged or trigger "
       "a security indication."

  Start Attribute
      Name = "commands"
      ID = 1
              Description = "commands and occurrences to be processed "
                    "by DMI2.0s SP for logging and/or security indications."
              Type = Start enum
                    0 = "unknown"
                    1 = "DmiRegister"
                    2 = "DmiUnregister"
                    3 = "DmiGetAttribute"
                    4 = "DmiSetAttribute"
                    5 = "DmiGetMultiple"
                    6 = "DmiSetMultiple"
                    7 = "DmiAddRow"
                    8 = "DmiDeleteRow"
                    9 = "DmiAddComponent"
                    10= "DmiAddLanguage"
                    11 = "DmiAddGroup"
                    12 = "DmiDeleteComponent"
                    13 = "DmiDeleteLanguage"
                    14 = "DmiDeleteGroup"
                    15 = "DmiRegisterCi"
                    16 = "DmiList"
                    17 = "Authentication Expired"
                    18 = "DmiOriginateEvent"
      End enum
      Access = Read-Only
      Storage = Common
      Value = "unknown"
  End Attribute

  Start Attribute
      Name = "level"
      ID = 2
              Description = "This command will be processed under the \n"
                    "specified condition. "
              Type = Start enum
                    0 = "unknown"
                    1 = "process if success"
                    2 = "process if security failure"
                    3 = "process if success or security failure"
                    4 = "process if non-security failure"
                    5 = "process if success or non-security failure"
                    6 = "process if security or non-security failure"
```

```
                    7 = "process if success or security failure or non-security failure"
               End enum
               Access= Read-Only
               Storage = Common
               Value = "unknown"
 End Attribute

        Start Attribute
            Name = "action"
            ID = 3
            Description = "The processing action to take."
            Type = Start enum
                  0 = "unknown"
                  1 = "log"
                  2 = "send security indication"
                  3 = "log and send security indication"
            End enum
            Access = Read-Only
            Storage = Common
            Value = 0
        End Attribute

        Start Attribute
     Name = "class string filter"
     ID = 4
              Type = String(256)
            Storage = Common
     Access = Read-Only
            Description = "The logging and/or security indication is performed \n"
                 "on groups whose class string matches the filter. \n"
                 "String || is a wildcard meaning all groups."
            Value = "||"
        End Attribute
End Group
```

For example, in order to log all the successful `DmiSetAttribute` commands, and log and generate a security indication upon all the modifications of the policy, the table should be set to:

```
Start Table
    Name = "DMI Logging Table"
    Class = "DMTF|SP Logging and Security Indication Characteristics|001"
    Id = 9

{ "DmiSetAttribute", "log", "process if success" }

{ "DmiAddRow", "log and send security indication", "process if success",
  "DMTF|POLICY_DB|" }

{ "DmiDeleteRow", "log and send security indication", "process if success",
  "DMTF|POLICY_DB|" }

End Table
```

12.3 AUTHENTICATION PROTOCOLS

A DMIv2.0s Service Provider may support one or more authentication protocols. For example, it may support authentication through NT login and through digital certificates. The `Authentication Protocols` group is a table instrumented by the DMIv2.0s Service Provider that lists all the authentication protocols supported along with their RPC type and transport type (since some authentication protocols may be supported only on some of the RPCs). The definition of attributes `SP RPC Type` and `SP Transport Type` are similar to those of attributes `Subscriber RPC Type` and `Subscriber Transport Type` in the `SP Indication Subscription` table.

A management application may list the rows of the `Authentication Protocols` table to find out which authentication protocols are supported by a DMIv2.0s Service Provider. It is recommended to set a policy that allows any role to read the authentication protocols table, so that it can be read by management applications without authenticating. That is, it is recommended that the policy contain the following row:

```
{"dmi_default", "DmiGetAttribute", "Allow", "DMTF|Authentication Protocols|", , , , }
```

The `Authentication Protocols` group is listed below:

```
Start Group
 Name = "Authentication protocols"
 Class = "DMTF|Authentication Protocols|001"
 Key = 1,2,3
 Description = "This table lists authentication protocols supported."

 Start Attribute
           Name = "Authentication Protocol Type "
           ID = 1
           Description = "This is an identifier of the type of Authentication "
                 "in use by the SDMI SP."
           Access = Read-Only
           Storage = Common
           Type = Start enum
       1 = "ONC UNIX"
       2 = "Kerberos"
       3 = "Windows NT4 Authentication"
       4 = "NetWare 4.1"
       5 = "X.509"
       6 = "DES"
           End Enum
 End Attribute

 Start Attribute
           Name = "SP RPC Type"
           ID = 2
           Description = "This is an identifier of the type of RPC in "
                 "use by the SP."
           Access = Read-Only
           Storage = Common
           Type = String(64)

           // NOTE:   the allowable RPC Type strings are
           //    "DCE RPC"
           //    "ONC RPC"
           //    "TI  RPC"
 End Attribute

 Start Attribute
           Name = "SP Transport Type"
           ID = 3
           Description = "This is an identifier of the type of Transport in "
                 "use by the SP."
           Access = Read-Only
           Storage = Common
           Type = String(64)

           // NOTE:   the allowable Transport Type strings are
```

```
                //    "ncacn_dnet_nsp"
                //    "ncacn_ip_tcp"
                //    "ncadg_ip_udp"
                //    "ncacn_nb_nb"
                //    "ncacn_nb_tcp"
                //    "ncacn_nb_ipx"
                //    "ncacn_np"
                //    "ncacn_spx"
                //    "ncadg_ipx"
                //    "ncalrpc"
  End Attribute
End Group
```

12.4 POLICY GROUP

The `Policy_DB` group is a tabular group in which each row specifies a group of ***DMI*** commands that can or cannot be performed on the system according to the role of the user invoking the command, the group's class string and attribute ID accessed by the command. To allow specifying different policies for different groups with the same class string, the value of an additional attribute can be specified, in which case the policy row applies only to those components that contain the specified attribute with the specified value. If one or more rows in the policy specify roles that can perform a command on a component/group/attribute, then only those roles specified will be allowed to perform that command; otherwise, all roles are allowed to. A more precise description of the authorization algorithm can be found in Section 13.2, and pseudo-code is listed in Section 13.6.

The value of some of the attributes in a policy row may be a wildcard. The syntax of wildcards is specified in the description of each attribute. Wildcards are used by the DMIv2.0s Service Provider when matching an incoming command against policy rows for authorization. The policy group definition is listed below.

```
Start Group
 Name = "DMI Policy"
 Class = "DMTF|Policy_DB|001"
 Key = 1,2,3,4,5,6,7,8
 Description = "This table contains the DMIv2.0s SP authorization policy."
```

12.4.1 Role

Attribute `role` in a policy row specifies the role that a row applies to. Roles names are encoded as strings. Role names are opaque to the DMIv2.0s Service Provider: the Service Provider matches the list of roles of a user against the policy in order to authorize each command.

```
Start Attribute
           Name = "Role"
           Id = 1
    Description = "Role to which this row applies."
           Storage = Specific
           Access = Read-Only
           Type = String(256)
           Value = ""
End Attribute
```

12.4.2 Command

Attribute `command` in a policy row specifies the command or group of commands that a row applies to. Note that all ***DMI*** listing commands are grouped together. Values out of range are reserved and should not be set.

```
Start Attribute
    Name = "Command"
    Id = 2
    Description = "Command to which this row applies."
    Storage = Common
    Access = Read-Only
    Type =  Start enum
       1 = "DmiGetAttribute"
       2 = "DmiSetAttribute"
       3 = "DmiAddRow"
       4 = "DmiDeleteRow"
       5 = "DmiAddGroup"
       6 = "DmiDeleteGroup"
       7 = "DmiAddComponent"
       8 = "DmiDeleteComponent"
       9 = "DmiAddLanguage"
       10 = "DmiDeleteLanguage"
       11 = "DmiList"
    End enum
End Attribute
```

The following commands are allowed to any role regardless of the policy: DmiRegister, DmiUnregister, DmiGetVersion, DmiGetConfig and DmiSetConfig.

A DmiSetMultiple command is allowed if each of the individual sets is allowed. In a DmiGetMultiple command, each individual get is authorized separately, and partial attribute data may be returned. See Section 18 for a precise description of the behavior of DmiGetMultiple in the presence of errors. Note that a DmiGetMultiple command that returns a key list (when RequestMode is DMI_FIRST or DMI_NEXT) requires DmiGetAttribute permission on each of the keys.

12.4.3 Authorization

Attribute `authorization` in a policy row specifies whether the row allows or denies the specified role to perform the specified command. The attribute `authorization` is of type `enum {"Deny", "Allow"}`. Values out of range are reserved and should not be used.

```
Start Attribute
    Name = "Authorization"
    Id = 3
    Description = "Defines whether this row allows or denies access."
    Storage = Common
    Access = Read-Only
    Type = Start enum
       0 = "Deny"
       1 = "Allow"
    End enum
End Attribute
```

Attributes 4 through 8 in a policy row specify the component/group/attribute that the policy row applies to. Not all of attributes 4 through 8 in a policy row are relevant to each command. For example, `AttributeID` is not relevant to `DmiAddComponent` commands. The policy attributes that are relevant to each command type are summarized in a table in Section 13.2.

12.4.4 Class

This attribute specifies the groups that a policy row applies to. The attribute `Class` is of type string. The semantics of this attribute is similar to that of the class string parameter to the `ListComponentsByClass` command in the Management Interface. Partial class strings may be specified. For example, the partial class string "`DMTF|Serial Ports|`" will match all DMTF defined versions of the standard serial port group.

```
Start Attribute
    Name = "Class"
    Id = 4
    Description = "Class filter of groups to which this row applies."
    Storage = Specific
    Access = Read-Only
    Type = String(256)
    Value = "||"
End Attribute
```

12.4.5 Attribute ID

`Attribute ID` specifies the attribute that a policy row applies to. The attribute `Attribute ID` is of type integer. A value of zero is a wildcard meaning that the policy row applies to all the attributes in the group specified by `Class`. This makes it easy to protect a whole group. When a tabular group is accessed, the policy row applies to attribute `Attribute ID` in all rows.

```
Start Attribute
    Name = "AttributeID"
    Id = 5
    Description = "Attribute ID to which this row applies. 0 is wildcard."
    Storage = Specific
    Access = Read-Only
```

```
    Type = Integer
    Value = 0
End Attribute
```

12.4.6 Additional Class, Attribute ID, Value

To narrow down the scope of a policy row, in case there is more than one group in the system with the same class string, specify an additional (class, attribute, value) triple. These attributes narrow down the scope of a policy row so that it does not apply to all the groups of class `Class`. `Class2` is a string, `Attribute ID2` is an integer, `Value2` is an octet string representing the value of an attribute with the same syntax as `<value statement>` in a MIF file. If `Class2` is an empty string, `Attribute ID2` and `Value2` are ignored and the policy row applies to all groups of class `Class`.

When a management application attempts to perform a command, the DMIv2.0s Service Provider checks if any rows in the policy apply to this command. Policy rows in which `Class2` is specified apply to a command only if the component being accessed contains a group whose class string is `Class2` and this group contains an attribute with attribute ID `Attribute ID2` whose value is equal to `Value2`.

If the group is a tabular group, the policy row applies if the value `Attribute ID2` is `Value2` in the first row.

```
Start Attribute
    Name = "Class2"
    Id = 6
    Description = "Narrow down the scope of this row to components that "
        "contain a group with this class in which attributeID2 has value2."
    Storage = Specific
    Access = Read-Only
    Type = String(256)
    Value = ""
End Attribute


Start Attribute
    Name = "AttributeID2"
    Id = 7
    Description = "Attribute whose value is used to narrow down the scope "
                  "of this policy row."
    Storage = Specific
    Access = Read-Only
    Type = Integer
    Value = 0
End Attribute


Start Attribute
   Name = "Value2"
   Id = 8
   Description = "Value used to narrow down the scope of this policy row."
            Storage = Specific
   Access = Read-Only
            Type = OctetString(1024)
   Value = ""
End Attribute
```

In the following example:

```
{"tester", "DmiSetAttribute" , "Allow", "DMTF|Network Adapter 802 Port|001", ,
    "DMTF|ComponentID|001", 1, "Intel" }
```

role "tester" is allowed to perform `DmiSetAttribute` on any attribute in a group whose class string is `"DMTF|Network Adapter 802 Port|001"` in a component whose manufacturer is `"Intel"` (that is, a component that contains a group whose class string is `"DMTF|ComponentID|001"` and the value of attribute number 1 in that group is `"Intel"`).

12.4.7 Example

Here’s an example of the authentication protocols and policy tables:

```
Start Table
  Name = "DMI Authentication Protocols Table"
  Class = "DMTF|Authentication Protocols|001"
  Id = 8

  {"Windows NT4 Authentication", "DCE RPC", "ncacn_ip_tcp"}
  {"DES", "ONC RPC", " ncadg_ip_udp"}
End Table


Start Table
  Name = "DMI Policy Table"
  Class = "DMTF|Policy_DB|001"
  Id = 7

  // allow role 'IT' to add and remove components
  {"IT", "DmiAddComponent", "Allow", , , , , }
  {"IT", "DmiDeleteComponent", "Allow", , , , , }

          // allow role 'helpdesk' to set attributes
          {"helpdesk", "DmiSetAttribute" , "Allow", , , , }
          // allow role "HW support" to configure temp probe
          {"HW support", "DmiSetAttribute" , "Allow", "DMTF|Temperature Probe|", , , , }
          // role "IBM support", not "helpdesk" takes care of IBM components
          {"IBM support", "DmiSetAttribute" , "Allow", "IBM||", , , , }
          {"helpdesk", "DmiSetAttribute" , "Deny", "IBM||", , , }
End Table
```

The policy table allows:

- role `"IT"` to add and delete components.
- role `'helpdesk'` to set the value of any attribute except those in groups whose class string contains `"IBM"` as defining body.
- role `"HW support"` to set the value of any attribute in the “Temperature Probe” group.
- role `"IBM support"` to set the value of any attribute in any group whose class string contains `"IBM"` as defining body.

12.5 SPECIAL DMIv2.0S ROLES

The authentication module is responsible for assigning a list of roles to a user upon management application registration. Although ***DMIv2.0s*** does not specify the mechanism for associating user identities with roles, the recommended mechanism is the operating system user groups or digital certificate attributes. ***DMIv2.0s*** defines a special role, `dmi_default`, that is assigned to every management application, including those that use a non-authenticated RPC. Therefore, commands that are permitted to role `dmi_default` are actually permitted to all users. For example, the following row in the policy allows all users to read the authentication protocols table:

```
{"dmi_default", "DmiGetAttribute", "Allow", "DMTF|Authentication Protocols|", , , , }
```

To ease the configuration of ***DMIv2.0s***, it is recommended that ***DMIv2.0s*** administrators define a role named `dmi_admin` and allow this role to perform ***DMI*** database management operations (such as component installation and removal) and to modify the policy. To implement this, the policy table would contain the following rows:

```
{"dmi_admin", "DmiAddGroup", "Allow", , , , , }
{"dmi_admin", "DmiDeleteGroup", "Allow", , , , , }
{"dmi_admin", "DmiAddComponent", "Allow", , , , , }
{"dmi_admin", "DmiDeleteComponent", "Allow", , , , , }
{"dmi_admin", "DmiAddRow", "Allow", "DMTF|POLICY_DB|001", , , , }
{"dmi_admin", "DmiDeleteRow", "Allow", "DMTF|POLICY_DB|001", , , , }
```

13. MANAGEMENT INTERFACE SECURITY

Management Interface security is the main feature of ***DMIv2.0s***. Management Interface security controls the access of management applications to ***DMI*** data and instrumentation.

Upon registration of a management application with the Service Provider, the Service Provider authenticates the management application, obtains the list of roles of the user invoking that management application and returns a management handle. Every subsequent ***DMI*** command requested through this management handle will be authorized by the DMIv2.0s Service Provider according to this list of roles and the policy.

Section 13.1 Authentication describes the interaction between the DMIv2.0s Service Provider and the underlying RPC authentication mechanism.

Section 13.2 Policy and authorization defines ***DMIv2.0s*** authorization of Management Interface commands issued by remote management applications.

Section 13.3 Policy protection, modification, and initialization discusses configuring the policy to control access to the policy itself, and lists the recommended initial policy.

Section 13.4 Indication subscription and delivery discusses security as it applies to the subscription of management applications for indications and delivery of those indications.

Section 13.5 Local management interface defines the security of the Management Interface when accessed directly by local management applications (rather than through an RPC).

13.1 AUTHENTICATION

DMIv2.0 uses Remote Procedure Call (RPC) standards for remoting the Management Interface. ***DMIv2.0s*** also uses RPC for authenticating the user of the management application. The RPC infrastructure on the RPC client (the management application) sends the identity of the user invoking the management application to the RPC infrastructure of the RPC server (the DMIv2.0s Service Provider). Upon registration of a management application, the DMIv2.0s Service Provider retrieves the identity of the user and extracts the associated roles list. The actual call used by the DMIv2.0s Service Provider to retrieve the identity of the user depends on the specific RPC being used (for example `rpc_binding_inq_auth_client()` on DCE RPC or `rq_cred` and `rqclntcred` in `struct svc_req` on ONC RPC). Optionally, the DMIv2.0s Service Provider may also perform authentication on subsequent Management Interface RPC calls, and verify that the identity of the caller is the identity of the caller of `DmiRegister`.

The name space of user identities depends on the specific RPC and operating system. For example, when using DCE RPC between Windows systems, user identities are of the form *host/name*, where *host* is the name of a Windows NT workstation, Windows NT server or NT domain, and *name* is the login name of a user. When using ONC between UNIX systems, the identity of a user is composed of its *uid* number.

The mapping of user identities onto roles is defined by the DMIv2.0s Service Provider implementation. This mapping may be a simple one-to-one mapping with each user identity being a role, or the role list may be contained in the user identity as, for example, an attribute in an X.509 certificate. It is recommended to use operating system groups to map users onto roles, since system administrators are already familiar with the concept of operating system user groups and with the tools used to manage their membership.

A management application may support more than one authentication protocol in order to manage several types of ***DMIv2.0s***-enabled computers. To select the proper authentication protocol for managing a specific computer, the management application can retrieve the list of authentication protocols supported by a DMIv2.0s Service Provider by retrieving the rows of the `Authentication Protocols` table. It is recommended that the policy configure this table to be readable by any role.

Certain authentication protocols implement the concept of expiration or revocation of an identity or of credentials. If such an authentication protocol is used, it is the responsibility of the RPC infrastructure to terminate the RPC session upon identity expiration or revocation. Subsequent commands attempted will fail with an error defined by the RPC infrastructure.

13.1.1 Non-authenticated registration

A management application may register with the DMIv2.0s Service Provider using `DmiRegister` but not perform the authentication protocol. This may be because the management application does not use authentication features of the RPC or because it uses an RPC that does not support authentication. In this case the DMIv2.0s Service Provider will assign a role list that contains only role `dmi_default` to the management application.

13.2 POLICY AND AUTHORIZATION

Authorization is the mechanism whereby the DMIv2.0s Service Provider decides whether a ***DMI*** command invoked by a user should be allowed or denied according to the command, its parameters, the user's roles, and the policy.

A role is said to be allowed to perform a given command if either:

- There is at least one row in the policy with `Authorization` equal to `"Allow"` that matches this role/command/parameters and there is no row in the policy with `Authorization` equal to `"Deny"` that matches this role/command/parameters.
- There is no row in the policy that matches the command/parameters.

When searching the policy for rows that match a command, the Service Provider checks only relevant policy attributes and command parameters. The command parameters and the policy attributes used for matching each command against the policy are listed in the following table. Note that for simplicity all ***DMI*** Listing commands have been grouped together, and can be allowed or denied to a role regardless of the component, group or attributes being listed.

Command	Command parameters checked for match	Policy attributes used for matching
`DmiGetAttribute`	Component, Group, Attribute	Class, AttributeID, Class2, AttributeID2, Value2
`DmiSetAttribute`	Component, Group, Attribute	Class, AttributeID, Class2, AttributeID2, Value2
`DmiDeleteRow`	Component, Group	Class, Class2, AttributeID2, Value2
`DmiAddRow`	Component, Group	Class, Class2, AttributeID2, Value2
`DmiDeleteGroup`	Component, Group	Class, Class2, AttributeID2, Value2
`DmiAddGroup`	Component	Class2, AttributeID2, Value2
`DmiDeleteComponent`	Component	Class2, AttributeID2, Value2
`DmiAddComponent`		
`DmiDeleteLanguage`	Component	Class2, AttributeID2, Value2
`DmiAddLanguage`	Component	Class2, AttributeID2, Value2
`DmiList`		

When a management application attempts to perform a command that requires authorization, the Service Provider searches the policy for rows that match the command. If there is no such row, then the command is allowed. If there are policy rows that match the command, the Service Provider checks whether one of the roles of the user invoking the command is allowed to perform the command, and allows or denies the command accordingly. Commands that a user is not authorized to perform are not performed and return with error `DMIERR_INSUFFICIENT_PRIVILEGES`. Pseudo-code for the authorization algorithm is listed in Section 13.6.

13.3 POLICY PROTECTION, MODIFICATION AND INITIALIZATION

The policy is stored as a tabular group in the MIF database. Access to the policy is controlled by the policy itself. For example, to allow role "dmi_admin" to modify the policy, the following rows should be included in the policy:

```
{"dmi_admin", "DmiAddRow" , "Allow", "DMTF|POLICY_DB|001", , , , }
{"dmi_admin", "DmiDeleteRow" , "Allow", "DMTF|POLICY_DB|001", , , , }
```

Roles other than "dmi_admin" will not be allowed to modify the policy, unless specifically allowed to by other policy rows.

Rows may be added to or removed from the policy table dynamically.

NOTE that attributes in the policy are read only, so the policy can be modified only by adding or deleting rows.

When the DMIv2.0s Service Provider is installed, it creates an initial default policy table specified by the Service Provider implementation. The recommended default policy is listed below, though the system manufacturer may chose to set a different policy at system initialization:

```
{"dmi_admin", "DmiAddComponent", "Allow", , , , , }
{"dmi_admin", "DmiDeleteComponent", "Allow", , , , , }
{"dmi_admin", "DmiAddGroup", "Allow", , , , , }
{"dmi_admin", "DmiDeleteGroup", "Allow", , , , , }
{"dmi_admin", "DmiAddRow", "Allow", "DMTF|POLICY_DB|001", , , , }
{"dmi_admin", "DmiDeleteRow", "Allow", "DMTF|POLICY_DB|001", , , , }
{"dmi_admin", "DmiAddRow", "Allow", "DMTF|SP Logging and Security Indication
  Characteristics|001", , , , }
{"dmi_admin", "DmiDeleteRow", "Allow", "DMTF|SP Logging and Security Indication
  Characteristics|001", , , , }
{"dmi_admin", "DmiAddRow", "Allow", , , , , }
{"dmi_admin", "DmiDeleteRow", "Allow", , , , , }
{"dmi_admin", "DmiSetAttribute", "Allow", , , , , }
{"dmi_default", "DmiAddRow", "Allow", "DMTF|SP Indication Subscription|001", , , , }
{"dmi_default", "DmiDeleteRow", "Allow", "DMTF|SP Indication Subscription|001", , , , }
{"dmi_default", "DmiAddRow", "Allow", "DMTF|SPFilterInformation|001", , , , }
{"dmi_default", "DmiDeleteRow", "Allow", "DMTF|SPFilterInformation|001", , , , }
{"dmi_default", "DmiGetAttribute", "Allow", , , "DMTF|ComponentID|001", 2, "Win32 DMI
  Service Provider"}
{"dmi_default", "DmiGetAttribute", "Allow", "DMTF|Authentication Protocols|", , , , }
```

13.4 INDICATION SUBSCRIPTION AND DELIVERY

This section reviews the mechanisms involved in indication subscription and delivery and their interaction with ***DMIv2.0s*** security. ***DMI*** management applications interested in receiving event notifications must subscribe for indications with the Service Provider. The Service Provider component includes two tabular groups through which a management application can subscribe for indications: `SP Indication Subscription` and `SP Filter Information`. Management applications subscribe for indications with the Service Provider by adding rows to these tables.

NOTE that subscribing for indications is different from performing ***DMI*** commands in two ways:

Indication subscription is persistent; that is, it stays in effect even after the end of the management session during which the subscription was performed.

Indications are initiated by the Service Provider and consumed by management applications (unlike ***DMI*** commands which are initiated by management applications and performed by the Service Provider).

The indication server block in the management application (Section 11.1) is actually an RPC server and the indication client block in the Service Provider acts as its RPC client. The indication subscription and filter tables are stored in the MIF database which is persistent across management sessions. The indication subscription table contains a list of managing nodes that have subscribed to receive indications, and information required to forward indications to them. When an indication is generated, the Service Provider looks up the subscription and filter tables, opens an RPC session to each of the subscribed event consumers that has set the appropriate filters, and sends the indication.

DMIv2.0s provides limited support for securing indication subscription and delivery because, in general, indications carry no sensitive data; they often carry no data at all. For example, when a temperature probe detects that a system's temperature is too high, it generates an event containing data identifying this particular probe group. Upon receiving the indication, the management application will query the current temperature of the system by invoking `DmiGetAttribute` on the appropriate attribute in the probe group and perform appropriate actions.

Indication subscription is protected by controlling access to the `SP Indication Subscription` and `SP Filter Information` tables through the policy. The policy can define which roles are allowed to add rows to these tables; other roles will not be able to subscribe. However, the RPC session opened by the Service Provider to deliver an indication to a management application is *not* authenticated.

13.5 LOCAL MANAGEMENT INTERFACE

The Management Interface defined by ***DMIv1*** is a local API. The Management Interface defined by ***DMIv2.0*** can be accessed remotely through a Remote Procedure Call mechanism. Note that management applications running on the managed system itself can also access ***DMIv2.0s*** through an RPC. Remote Procedure Calls within one system can be performed through a special local RPC transport (for example `ncalrpc`) or through a networking RPC transport (for example, `ncacn_ip_tcp`) using the managed system's address or a loopback address as node address. In the context of this specification, invoking the Management Interface through an RPC from the same system on which the Service Provider is running is *not* considered a local access, and the access control mechanism applied is the same as when the Management Interface is invoked through an RPC from a different system, as defined in the previous sections.

The Management Interface defined by ***DMIv2.0*** can also be accessed through a local interface within the managed system. This interface is usually a well known entry point in a DLL or a system call. This section defines security as it applies to management applications that access the DMIv2.0s Service Provider through a local API, be it the ***DMIv1*** Management Interface or the ***DMIv2*** Management Interface. The behavior of the DMIv2.0s Service Provider with local management applications is controlled by attribute `enable local security` in the `SP Characteristics` group. If the value of this attribute is `True` when the DMIv2.0s Service Provider initializes, local management application security applies. Otherwise, all local management applications have unlimited access to the Management Interface. The security mechanisms applied by ***DMIv2.0s*** to local management applications are a simplified form of the mechanisms defined for remote management applications:

- Authentication is binary according to whether the local management application is invoked by a privileged user or not (see Section 10.2.6 for a definition of privileged users).
- Authorization is binary: local management applications invoked by a privileged user are allowed to perform any ***DMI*** command, whereas those invoked by a non-privileged user are not allowed to access ***DMI***.
- Indication subscription and delivery are affected accordingly: local management applications invoked by a privileged user may subscribe for and receive indications, whereas those invoked by a non-privileged user may not.

13.5.1 Caveat: component instrumentation registration as a local management application

Component instrumentation often registers through the local Management Interface in order to access ***DMI*** information. For example, component instrumentation can use ***DMI*** information to find out the component ID of the component it instruments, or to discriminate between two instances of the same component installed on the system, or to store data pertaining to the component instrumentation. If local management application security is enabled and component instrumentation registers as a local management application through the local ***DMI*** API, the security mechanisms described in Section 13.5 apply. Therefore, if attribute `enable local security` is `True` when the DMIv2.0s Service Provider initializes, component instrumentation should be configured to run as privileged process in order to be able to use the Management Interface. See also Section 14 on component interface security.

13.6 AUTHORIZATION ALGORITHM PSEUDO-CODE

When searching the policy for rows that match a command, relevant command parameters are checked against each policy row's attributes according to the table in Section 13.2. A fully specified policy row `{Role, Cmd, Authz, Class1, AttrId1, Class2, AttrId2, Value2}` is said to match a ***DMI*** command with parameters CID, GID, AID if:

The class string of group GID matches the class filter `Class1`.

AID is `AttrId1`.

Component CID contains a group whose class string is `Class2` and an attribute in that group whose ID is `AttrId2` and whose value is `Value2`.

Pseudo code for the authorization algorithm follows:

```
if (this command is DmiRegister, DmiUnregister, DmiGetVersion, DmiGetConfig or
    DmiSetConfig) then
        return allowed
else if (this command is DmiSetMultiple) then
    if (each of the sets is allowed per this algorithm) then
        return allowed
    else
        return denied
else if (there are policy rows that match this command) then {
    for (each role R of this user) {
        if (there is a policy row matching this command such that role=R and auth=deny) then
            continue /* for */
        if (there is a policy row matching this command such that role=R and auth=allow) then
            return allowed
    } /* for */
    return denied
}
else return allowed
```

14. COMPONENT INTERFACE SECURITY

The main objective of ***DMIv2.0s*** is to control access of managed systems by remote management applications. Nonetheless, ***DMIv2.0s*** also provides features to control registration of component instrumentation and protect the system from software that behaves like a component instrumentation but is not a legitimate component instrumentation. This section defines security as it applies to component instrumentation interfacing with the DMIv2.0s Service Provider, be it DMIv1 component instrumentation or DMIv2 component instrumentation. Component Interface security is controlled by attribute `enable local security` in the `SP Characteristics` group. If the value of this attribute is `True` when the DMIv2.0s Service Provider initializes, Component Interface security applies. Otherwise, access to the DMIv2.0s Component Interface is unrestricted.

DMI defines two types of interface between the Service Provider and component instrumentation: direct and overlay. Instrumentation using the overlay interface is declared in the MIF by a value statement of the form `value = *"name"`, where `name` has been previously defined in a **path** definition within the component definition. Upon a `DmiGetAttribute` or `DmiSetAttribute` to this attribute, the Service Provider loads and invokes the code located in the file corresponding to the **path** definition for the OS running on the managed system. **The overlay Component Interface is not supported by *DMIv2.0s*.**

Instrumentation using the direct interface must register with the DMI Service Provider when it wishes to notify the Service Provider of its immediate availability. (Attributes instrumented through the direct interface and static attributes whose value is stored in the MIF database are defined in the same way in the MIF.) Upon registration, direct interface instrumentation provides the Service Provider with entry points through which the Service Provider can later invoke the instrumentation. The mechanics of "connecting" to the DMI Service Provider to register or issue commands may differ among operating systems and DMI Service Provider implementations.

If attribute `enable local security` is `True` when the DMIv2.0s Service Provider initializes, registration of component instrumentation will succeed only if the component instrumentation is a privileged process as defined in Section 10.2.6. That is, invocations of `DmiRegisterCi` by a non-privileged process will fail and return `DMIERR_INSUFFICIENT_PRIVILEGES`.

Registration of direct interface component instrumentation for an attribute overrides the previous access mechanism for the attribute, which could be a static value in the MIF database, an overlay, or a previous registration. In ***DMIv2.0s***, this behavior can be controlled through attribute `disable CI override` in the `SP Characteristics` group. If the value of this attribute is `True` when the DMIv2.0s Service Provider initializes, invocations of `DmiRegisterCi` which would override a previous direct interface instrumentation registration will fail and return `DMIERR_INSUFFICIENT_PRIVILEGES`. See also Sections 16 and 17 on notifications generated as a result of instrumentation override.

15. MIF DATABASE PROTECTION

The MIF database is local to the managed PC. Since the policy is stored in the MIF database, it is necessary to protect the database. The contents of the database are protected from unauthorized access by ***DMI*** management applications through the Management Interface security, based on roles and policy for remote management applications and on operating system privileged processes for local management applications. However, it is also necessary to protect the database in its stored form. The DMIv2.0s Service Provider uses operating system or file system mechanisms to protect the MIF database, if such mechanisms are available. The DMIv2.0s Service Provider will set up the ACL of the MIF database file such that only privileged processes can read, write or erase the MIF database.

16. SECURITY INDICATIONS

This section describes security indications to be sent to monitoring management applications. To avoid generating spurious indication traffic on the network, security indications are configurable. Security indications are declared in the DMIv2.0s Service Provider MIF with a standard event generation template group. The event generation group and the attributes sent in the indication block are described in Section 16.1. Security indication generation is controlled by the `SP Logging and Security Indication Characteristics` group see the definition of this group in Section 12.2. This group also controls the configuration of ***DMIv2.0s*** logging.

16.1 SECURITY INDICATION DATA

When an indication is delivered to a consumer, the data supplied includes a standard event generation group that is common to all standard events, and additional data that is specific to the event. Refer to Section 5 for the exact layout of the data in the indication data structure. The event generator group specifies the type of the event, the severity, the group associated with the component that generated the event, the system and subsystem concerned by the event. The event generation group is formatted according to the standard template in Section 16.1.1. Additional attributes are described in Section 16.1.2.

16.1.1 Security indication event generation group

```
Start Group
        Name = "Event Generation"
        Class = "EventGeneration|DMTF^^Security Indication|001"
        ID =
        Key = 5

        Start Attribute
                Name = "Event Type"
                ID = 1
                Description="The type of the event - This is actually "
                "the command which\ncaused this event to be generated."
                Type = Start Enum
                     0x00000 = "unknown"
                     0x00001 = "DmiRegister"
                     0x00002 = "DmiUnregister”
                     0x00003 = "DmiGetAttribute"
                     0x00004 = "DmiSetAttribute"
                     0x00005 = "DmiGetMultiple"
                     0x00006 = "DmiSetMultiple”
                     0x00007 = "DmiAddRow”
                     0x00008 = "DmiDeleteRow"
                     0x00009 = "DmiAddComponent"
                     0x0000A = "DmiAddLanguage"
                     0x0000B = "DmiAddGroup"
                     0x0000C = "DmiDeleteComponent"
                     0x0000D = "DmiDeleteLanguage"
                     0x0000E = "DmiDeleteGroup"
                     0x0000F = "DmiRegisterCi"
                     0x00010 = "DmiListComponents"
                     0x00011 = "DmiListComponentsByClass"
                     0x00012 = "DmiListLanguages"
                     0x00013 = "DmiListClassNames"
                     0x00014 = "DmiListGroups"
                     0x00015 = "DmiListAttributes"
                     0x00016 = "Authentication Expired"
                     0x00017 = "DmiOriginateEvent"
                End Enum
                Access = Read-Only
                Storage = Common
```

```
        Value = "unknown"
End Attribute


Start Attribute
        Name = "Event Severity"
        ID = 2
        Description = "The severity of this event."
        Type = Start Enumeration
               0x001 = "Monitor"
               0x002 = "Information"
               0x004 = "OK"
               0x008 = "Non-Critical"
               0x010 = "Critical"
               0x020 = "Non-Recoverable"
               End Enumeration
        Access = Read-Only
        Storage = Specific
        Value = "Information"
End Attribute


Start Attribute
        Name = "Is Event State-Based?"
        ID = 3
        Description = "The value of this attribute "
                "determines whether the Event being reported "
                "is a state-based Event or not. If the value "
                "of this attribute is TRUE then the Event is "
                "state-based. Otherwise the Event is not "
                "state-based."
        Type = "BOOL"
        Access = Read-Only
        Storage = Specific
        Value = "False"
End Attribute


Start Attribute
        Name = "Event State Key"
        ID = 4
        Description = "A unique, single integer key into the"
              " Event State group if this is a state-based"
        " Event. If this is not a state-based Event then"        "
this attribute's value is not defined."
        Type = Integer
        Access = Read-Only
        Storage = Common
        Value = 0 // ignored since event is not state-based //
End Attribute


Start Attribute
        Name = "Associated Group"
        ID = 5
        Description = "The class name of the group that is"
        " associated with the events defined in this"
              " Event Generation group."
        Type = String
        Access = Read-Only
        Storage = Common
        Value = "DMTF|SP Logging and Security Indication
              Characteristics|001"
End Attribute


Start Attribute
        Name = "Event System"
        ID = 6
        Description = "The major functional aspect of the "
```

```
                "product causing the fault."
                Type = Start enum
                     0x000 = "SP"
                End enum
                Access = Read-Only
                Storage = Specific
                Value = 0 // value to be filled in by instrumentation//
        End Attribute


        Start Attribute
                Name = "Event Subsystem"
                ID = 7
                Description = "The minor functional aspect of the"
                     " product causing the fault."
                Type = Start enum
                     0x000 = "SP"
                End enum
                Access = Read-Only
                Storage = Specific
                Value = 0 // value to be filled in by instrumentation//
        End Attribute
End Group
```

The values of attributes in the event generation group are filled by the instrumentation (which in this case is part of the DMIv2.0s Service Provider itself) according to the specific security indication. The associated group class string is `"DMTF|SP Logging and Security Indication Characteristics|001"` which is the class string of the corresponding configuration group; the event system and event subsystem attributes will be set to zero. Additional information for each security indication is provided in the additional attributes defined in Section 16.1.2. Optionally, ***DMIv2.0s*** Service Provider implementations may provide four optional attributes in the event generation group. These attributes are defined in section 3.2.2.3.

16.1.2 Security indication additional attributes

Additional attributes include information about the management application that performed or attempted to perform an operation, the component, group, and attribute associated with the operation, the operation completion code and the level which caused the indication. Additional attributes are located in the fourth `DmiRowData` structure of the indication data structure. The values are formatted according to the following attribute definitions. The semantics of each attribute for each security indication type are specified at the end of this section.

```
Start Attribute
        Name = "Principal RPC Type"
        ID = 1
        Description = "This is an identifier of the type of RPC in use by the
        principal."
        Access = Read-Write
        Storage = Common
        Type = String(64)

        // NOTE: RPC strings are defined as follows
        //       "DCE"
        //       "ONC"
        //       "TI"
        Value = unknown
End Attribute


Start Attribute
        Name = "Principal Transport Type"
        ID = 2
        Description = "This is an identifier of the type of Transport in use by the
        Principal."
        Access = Read-Write
        Storage = Common
        Type = String(64)
        Value = unknown
```

```
        // NOTE:  the allowable Transport Type strings are
        //        "ncacn_dnet_nsp"
        //        "ncacn_ip_tcp"
        //        "ncadg_ip_udp"
        //        "ncacn_nb_nb"
        //        "ncacn_nb_tcp"
        //        "ncacn_nb_ipx"
        //        "ncacn_np"
        //        "ncacn_spx"
        //        "ncadg_ipx"
        //        "ncalrpc"
End Attribute


Start Attribute
        Name = "Principal Addressing"
        ID = 3
        Description = "This is an identifier of the addressing information"
                  " in use by the Principal."
        Access = Read-Write
        Storage = Specific
        Type = String(1024)
        Value = unknown
End Attribute

Start Attribute
        Name = "Principal Id"
        ID = 4
        Access = Read-Write
        Storage = Specific
        Type = String(1024)
        Value = unknown
End Attribute

Start Attribute
        Name = "Component Id"
        ID = 5
        Description = "This is the Id of the component affected by the"
                  " operation performed or attempted."
        Access = Read-Write
        Storage = Common
        Type = Integer
End Attribute

Start Attribute
        Name = "Group Id"
        ID = 6
        Description = "This is the Id of the group affected by the operation"
                  " performed or attempted."
        Access = Read-Write
        Storage = Common
        Type = Integer
End Attribute


Start Attribute
        Name = "Attribute Id"
        ID = 7
        Description = "This is the Id of the attribute affected by the operation"
                  " performed or attempted."
        Access = Read-Write
        Storage = Common
        Type = Integer
End Attribute


Start Attribute
        Name = "Level"
        ID = 8
        Description = "This is the actual level that caused the indication."
```

```
        Access = Read-Write
        Storage = Common
        Type = Start Enumeration
                 0x000 = "Unknown"
                 0x001 = "Success"
                 0x002 = "Security Failure"
                 0x004 = "Non-Security Failure"
         End Enumeration
End Attribute

Start Attribute
        Name = "Completion Code"
        ID = 9
        Description = "This is the error code the command completed with."
        Access = Read-Write
        Storage = Common
        Type = Integer
End Attribute
```

The value of the additional attributes is defined as follows:

`Principal RPC type`, `Principal Transport Type`, `Principal Addressing` and `Principal ID` identify the remote management application performing or attempting to perform the operation that caused the security indication. Their definition is similar to that of the corresponding attributes in the ***DMI*** indication subscription table except for `Principal ID`. `Principal ID` is the name of the user invoking the remote management application. If the name of the user cannot be obtained by the Service Provider, `Principal ID` will be a number identifying the user or the remote management application (such as a UNIX user ID or a NetWare NLM ID).

If the security indication is triggered by a component instrumentation or local management application, `Principal RPC Type` will be `"local"`, `Principal Transport Type` will be `"dmi"`, and `Principal Addressing` will be an empty string.

The next three attributes are component, group and attribute Id input parameters of the command that triggered the security indication, or zero for parameters not specified by the command (for example, `DmiRegister` and `Authentication Expired` have no associated component, group nor attribute.). If the command is `DmiSetMultiple`, `DmiGetMultiple`, or `DmiRegisterCI`, then the component/group/attribute that caused the security indication is returned in the indication block.

The next two attributes are the level that triggered the indication and the command completion code.

17. LOGGING

This section describes security logging entries logged by the DMIv2.0s Service Provider for future retrieval by monitoring applications at their convenience. The logging mechanism is similar to the security indications mechanisms described in Section 16: the information logged is similar to the information that is included in security indications.

Security logging is controlled by the `Service Provider Logging and Security Indication Characteristics` group. The first attribute `commands` determines which commands are to be logged. The second attribute `level` determines under what success/failure conditions the command is to be logged. The third attribute `action` determines whether to do logging, security indication or both. The fourth attribute `class string filter` provides the ability to filter for what groups the logging is done. See Section 12.2 for detailed description of the group.

The mechanism used to log the information is implementation-specific. It is recommended that DMIv2.0s Service Provider implementations use mechanisms provided by the operating system for logging, such as the NT event log on Windows NT, syslog on UNIX, or AUDITCON on NetWare. Tools for browsing log entries and configuring the maximum log size are usually provided. DMIv2.0s Service Provider implementations may define additional attributes to configure the logging mechanism, by, for example, providing the name of a log file or the address of a central system on which a consolidated log is maintained.

17.1 LOGGING INTERFACE

The Logging Interface is implemented by the logging module of the DMIv2.0s Service Provider. When this interface is invoked, the logging module adds an entry to the log. It is the Service Provider's responsibility to recognize when a command is to be logged and to call the interface provided by the logging module for each such command. In the case of `GetMultiple` and `SetMultiple`, the Service Provider will call the interface once for each element in the command that is to be logged (so, if the "level" attribute specifies that `SetMultiple` is to be logged always, and a number of attributes were successfully set by this command, then there will be a separate entry in the log for each attribute that was set). The interface provided is `DmiGenerateLog`.

17.1.1 DmiGenerateLog

```
DmiBoolean_t DmiGenerateLog (DmiLogInfo_t *info);
```

The one parameter is a pointer to a structure that contains all the information necessary to log the command. The definition type `DmiLogInfo_t` will be included in the `DMILOG.H` header file.

```
typedef struct DmiLogInfo {
    DmiCommandCode_t commandCode;
    DmiErrorStatus_t completionStatus;
    DmiString_t *componentName;
    DmiId_t componentId;
    DmiString_t *groupName;
    DmiId_t groupId;
    DmiString_t *attributeName;
    DmiId_t attributeId;
    DWORD logLevel,
    DmiString_t *rpcType;
    DmiString_t *transport;
    DmiString_t *address;
    DmiString_t *userNameorId;
    DmiString_t *impSpecificInfo;
} DmiLogInfo_t;
```

The definition of type `DmiCommandCode_t` will be included in the `DMILOG.H` header file. The constants for each command are as defined on page 235, with the addition of `DmiCiRegisterCode`, `DmiCiUnregisterCode` and `DmiOriginateEvent`.

```
typedef enum DmiCommandCode {
  DmiRegisterCode =         0x200,
  DmiUnregisterCode =       0x201,
  ...
  DmiGetattributeCode =     0x215,
  DmiSetattributeCode =     0x216,
  DmiCiRegisterCode =       0x220,
  DmiCiUnregisterCode =     0x221,
  DmiOriginateEvent =       0x222
} DmiCommandCode_t;
```

FIELD NAME	DIRECTION	DESCRIPTION
commandCode	In	An enumeration that identifies what the command is as defined above.
completionCode	In	The ***DMI*** status with which the command completed.
componentName	In	The name of the component that was referenced. NULL if not applicable.
componentId	In	The id of the component that was referenced. 0 if not applicable
groupName	In	The name of the group that was referenced. NULL if not applicable.
groupId	In	The id of the group that was referenced. 0 if not applicable
attributeName	In	The name of the attribute that was referenced. NULL if not applicable.
attributeId	In	The id of the attribute that was referenced. 0 if not applicable
logLevel	In	The actual level that caused the log.
rpcType	In	The name of the RPC that was used to deliver the command.
transport	In	The name of the transport that was used to deliver the command.
address	In	The address of the management application from which the command arrived. The format of this address depends on the transport used, and may be in numerical form.
userNameOrId	In	The name of the user that originated the command. Or the OS specific identifier of the process/application that originated the command, represented as an ASCII string.
impSpecificInfo	In	Implementation specific information that may be used.

18. DMIv2.0 AND DMIv2.0s COMPATIBILITY CONSIDERATIONS

This section discusses the interoperability of existing ***DMI*** management applications and component instrumentation with new DMIv2.0s Service Providers by summarizing relevant features introduced by the ***DMIv2.0s*** specification.

If the value of attributes `enable local security` and `disable CI override` are `False` when the DMIv2.0s Service Provider initializes, the local interface is fully compatible to that ***DMIv2.0***, and component instrumentation will run unchanged with the DMIv2.0s Service Provider, even if it does not run in the context of a privileged process.

If attribute `disable CI override` is `True` when the Service Provider initializes, component instrumentation attempting to register for an attribute for which component instrumentation has already registered will fail, returning error `DMIERR_INSUFFICIENT_PRIVILEGES`.

If the value of attribute `enable local security` is `True` when the DMIv2.0s Service Provider initializes, local component instrumentations and management applications that do not run in the context of a privileged process will not be able to interact with the DMIv2.0s Service Provider. `DmiRegisterCi` and `DmiRegister` will fail with error `DMIERR_INSUFFICIENT_PRIVILEGES`.

Management applications that register with the Service Provider using a non-authenticated RPC will be allowed to perform only commands that are allowed to role `dmi_default`.

A DMIv2.0s Service Provider returns the same result as a DMIv2.0 Service Provider for allowed commands.

For denied commands, a DMIv2.0s Service Provider returns error `DMIERR_INSUFFICIENT_PRIVILEGES`, whereas a DMIv2.0 Service Provider returns the command's result.

NOTE that a policy that contains no rows will allow any role to perform any command.

It may be possible to upgrade existing management applications that access the DMIv2.0 Service Provider through a non-authenticated RPC to ***DMIv2.0s*** by replacing the "front-end" module that interfaces with the RPC layer with a "front-end" that uses an authenticated RPC. Once the RPC has been replaced with an authenticated RPC, ***DMI*** commands sent by the management application will be authorized according to the policy and the identity of the user invoking the management application.

The behavior of `DmiGetMultiple` in the presence of errors, as described in the DMI2.0 Errata #1, is extended as follows:

When `DmiGetMultiple` is called without an attribute list, the DMIv2.0s Service Provider attempts to return all attributes in the group or row. Attributes that are `UNSUPPORTED`, `WRITE-ONLY` or that the management application is not authorized to get are omitted from the reply data. If a different error occurs when the Service Provider attempts to get an attribute, the Service Provider stops processing the request and returns data for all attributes up to, but not including, the attribute causing the error.

When `DmiGetMultiple` is called with a specific attribute list, any error that occurs when the Service Provider attempts to get an attribute causes the Service Provider to stop processing the request and return data for all attributes up to, but not including, the attribute causing the error.

If the Service Provider stops processing on the first attribute of a request, the Service Provider returns no data and a status according to the specific error (e.g. `DMIERR_ATTRIBUTE_NOT_SUPPORTED`, `DMIERR_ILLEGAL_TO_GET` or `DMIERR_INSUFFICIENT_PRIVILEGES` for an `UNSUPPORTED` attribute, a `WRITE ONLY` attribute or an attribute that the management application is not authorized to get, respectively).

If partial attribute data is returned, the operation's return status is `DMIERR_NO_ERROR_MORE_DATA`. When `DmiGetMultiple` returns a status of `DMIERR_NO_ERROR_MORE_DATA`, the caller should reissue the operation with a new attribute list. This new attribute list should start with the first attribute *not* returned in the previous call, and should contain all subsequent attributes from the original request.

APPENDIX A - ERROR CODES

Status codes are 32 bit unsigned values.

The error codes returned by an operating system are not passed back to a management application; the service provider maps operating system errors into its error range. The intent is to insulate management applications from operating system details.

Because the OS-related error codes are specific to a particular environment, they are not listed in this specification. Likewise, error codes from components are not listed here, but rather in the component MIF file.

Service Provider Error Codes

SYMBOL	VALUE	DESCRIPTION
DMIERR_ATTRIBUTE_NOT_FOUND	0x00100	Attribute not found
DMIERR_VALUE_EXCEEDS_MAXSIZE	0x00101	Value exceeds maximum size
DMIERR_COMPONENT_NOT_FOUND	0x00102	Component ID is not found
DMIERR_ENUM_ERROR	0x00103	Enumeration error
DMIERR_GROUP_NOT_FOUND	0x00104	Group not found
DMIERR_ILLEGAL_KEYS	0x00105	Illegal keys
DMIERR_ILLEGAL_TO_SET	0x00106	Illegal to set
DMIERR_OVERLAY_NAME_NOT_FOUND	0x00107	Component instrumentation not found
DMIERR_ILLEGAL_TO_GET	0x00108	Illegal to get
DMIERR_ROW_NOT_FOUND	0x0010a	Row not found
DMIERR_DIRECT_INTERFACE_NOT_REGISTERED	0x0010b	Direct interface not registered
DMIERR_DATABASE_CORRUPT	0x0010c	MIF database is corrupt
DMIERR_ATTRIBUTE_NOT_SUPPORTED	0x0010d	Attribute is not supported
DMIERR_VALUE_UNKNOWN	0x0010f	Value for this attribute is not known
DMIERR_BUFFER_FULL	0x00200	Buffer full
DMIERR_ILL_FORMED_COMMAND	0x00201	Ill-formed command
DMIERR_ILLEGAL_COMMAND	0x00202	Illegal command
DMIERR_ILLEGAL_HANDLE	0x00203	Illegal handle
DMIERR_OUT_OF_MEMORY	0x00204	Out of memory
DMIERR_NULL_COMPLETION_FUNCTION	0x00205	No confirm function
DMIERR_NULL_RESPONSE_BUFFER	0x00206	No response buffer
DMIERR_CMD_HANDLE_IN_USE	0x00207	Command handle is already in use
DMIERR_ILLEGAL_DMI_LEVEL	0x00208	DMI version mismatch
DMIERR_UNKNOWN_CI_REGISTRY	0x00209	Unknown CI registry
DMIERR_COMMAND_CANCELED	0x0020a	Command has been canceled
DMIERR_INSUFFICIENT_PRIVILEGES	0x0020b	Insufficient privileges
DMIERR_NULL_ACCESS_FUNCTION	0x0020c	No access function provided
DMIERR_FILE_ERROR	0x0020d	OS File I/O error
DMIERR_EXEC_FAILURE	0x0020e	Could not spawn a new task
DMIERR_BAD_SCHEMA_DESCRIPTION_FILE	0x0020f	Ill-formed SCHEMA
DMIERR_INVALID_FILE_TYPE	0x00210	Invalid file type
DMIERR_SP_INACTIVE	0x00211	Service provider is inactive
DMIERR_CANT_UNINSTALL_SP_COMPONENT	0x00213	Unable to remove the service provider component
DMIERR_NULL_CANCEL_FUNCTION	0x00214	No cancel function provided
DMIERR_INVALID_POOL	0x00215	Memory Pool handle is invalid
DMIERR_INVALID_PTR	0x00216	A memory Ptr passed was invalid
DMIERR_NO_POOL	0x00217	A memory pool is required for use with this function
DMIERR_FILE_TYPE_NOT_SUPPORTED	0x00218	The passed file type, while legal, is not supported by this implementation
DMIERR_CANT_UNINSTALL_COMPONENT_LANGUAGE	0x00219	Unable to install a components language mapping
DMIERR_CANT_UNINSTALL_GROUP	0x0021a	Unable to install the group
DMIERR_UNABLE_TO_ADD_ROW	0x0021b	The add row failed due to either a database problem or a component limitation
DMIERR_UNABLE_TO_DELETE_ROW	0x0021c	The delete row failed, due to either database problem or a component limitation

Non-Error Condition Codes

SYMBOL	VALUE	DESCRIPTION
DMIERR_NO_ERROR	0x00000	Success
DMIERR_NO_ERROR_MORE_DATA	0x00001	More data is available
DMIERR_DEFAULT_LANGUAGE_RETURNED	0x00002	The item requested did not have a language mapping installted that matched the one requested. The value was returned using the default language

APPENDIX B - DCE RPC IDL

DCE RPC PROGRAMMING FOR DMI 2.0

This section describes the process of creating a DCE RPC client-server application, such as we have with the DMI 2.0 Management Interface. In our case, the DMI 2.0 Service Provider is an RPC server and the management application is an RPC client. Most people reading this specification will be creating RPC clients.

There are three main steps involved in creating a client-server application: defining the interface, implementing the server, and implementing the client.

The Distributed Management Task Force has specified the DMI 2.0 interface in this document, and has created its formal description. This description is presented in the DCE Interface Description Language (IDL).

In the following sections, we will see that the IDL is used by both client and server developers when implementing their respective pieces of the application.

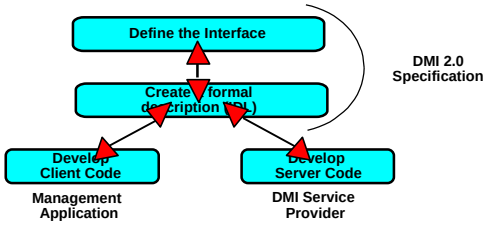


Figure B-1. Developing An RPC Application

CREATING THE DMI SERVICE PROVIDER

As a DMI Service Provider, you will develop the RPC server functionality for the DMI 2.0 interface.

The first step in this process (see Figure B-2) is to create the server stub code and interface header file. The stub code contains the actual routines that interface to the network software. The header file contains the data type declarations and function prototypes that you must implement.

To create the stub code, you compile the DMTF-supplied IDL, along with optional configuration information contained in the server ACF file. The IDL compiler is supplied as part of the RPC development environment for the Service Provider's platform. The ACF file allows you to tailor some aspects of the stub code generator. For example, does the generated stub code allocate memory on its stack, or on a heap? **Note**: The full set of ACF options are described in the various DCE RPC references.

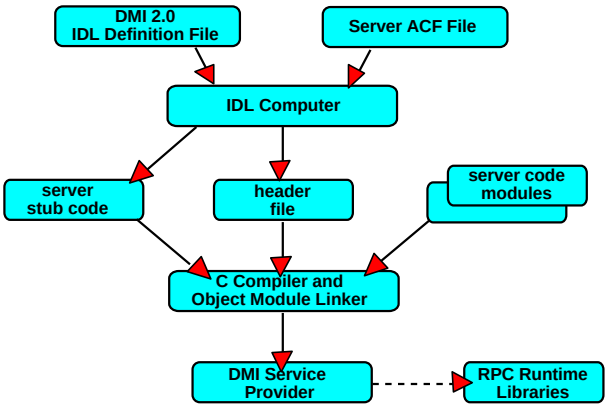


Figure B-2. Creating the DMI Service Provider.

After creating the stub code and header file, you must then write code to implement each of the application's entry points. In the DMI 2.0 case, this means you will write code for each MI function: DmiRegister, DmiUnregister, DmiListComponents, etc. Once this is done, your code and the server stub code are linked to produce the RPC server.

In addition to implementing the DMI 2.0 interface functions, you will need to write some code to register your server interface with the RPC runtime service, and to listen for incoming procedure calls from DMI 2.0 clients. A full description of the registration process is beyond the scope of this specification, but a small example may give some flavor as to what is involved.

In the following sample code, the DMI Service Provider obtains a dynamic endpoint from the system's endpoint mapper, registers the DMI interface (dmi_server_v2_0_s_ifspec), then listens for incoming procedure calls arriving on the connection-oriented TCP/IP protocol.

```
unsigned32             status;
unsigned char *        pszProtocolSequence = "ncacn_ip_tcp";
unsigned int           cMaxCalls           = 20;
rpc_binding_vector_p_t pbvBindings         = NULL;

// Initialize the RPC bindings and listen for requests.  No
// explicit endpoint is specified, so use the protocol sequence
// and register the endpoint with the endpoint mapper.  The string
// value of ncacn_ip_tcp says to use TCP/IP as the RPC transport.

rpc_server_use_protseq (pszProtocolSequence, cMaxCalls, &status);
check ("rpc_server_use_protseq", status);

rpc_server_inq_bindings (&pbvBindings, &status);
check ("rpc_server_inq_bindings", status);

rpc_ep_register (dmi_server_v2_0_s_ifspec, pbvBindings, 0, 0, &status);
check ("rpc_ep_unregister", status);

rpc_server_register_if (dmi_server_v2_0_s_ifspec, 0, 0, &status);
check ("rpc_server_register_if", status);

rpc_server_listen (cMaxCalls, &status);
check ("rpc_server_listen", status);

// When the rpc_server_listen() function returns, we are done
// listening so unregister our interface and exit.

rpc_server_unregister_if (dmi_server_v2_0_s_ifspec, 0, &status);
check ("rpc_server_unregister_if", status);

rpc_ep_unregister (dmi_server_v2_0_s_ifspec, pbvBindings, 0, &status);
```

```
check ("rpc_ep_unregister", status);

rpc_binding_vector_free (&pbvBindings, &status);
check ("rpc_binding_vector_free", status);
```

CREATING THE MANAGEMENT APPLICATION

As a DMI management application writer, you will be developing an RPC client. The development process for RPC clients is very similar to that of RPC servers. The differences are that you will be linking against the RPC client stubs instead of the server stubs, and you will be calling the interface functions instead of implementing them.

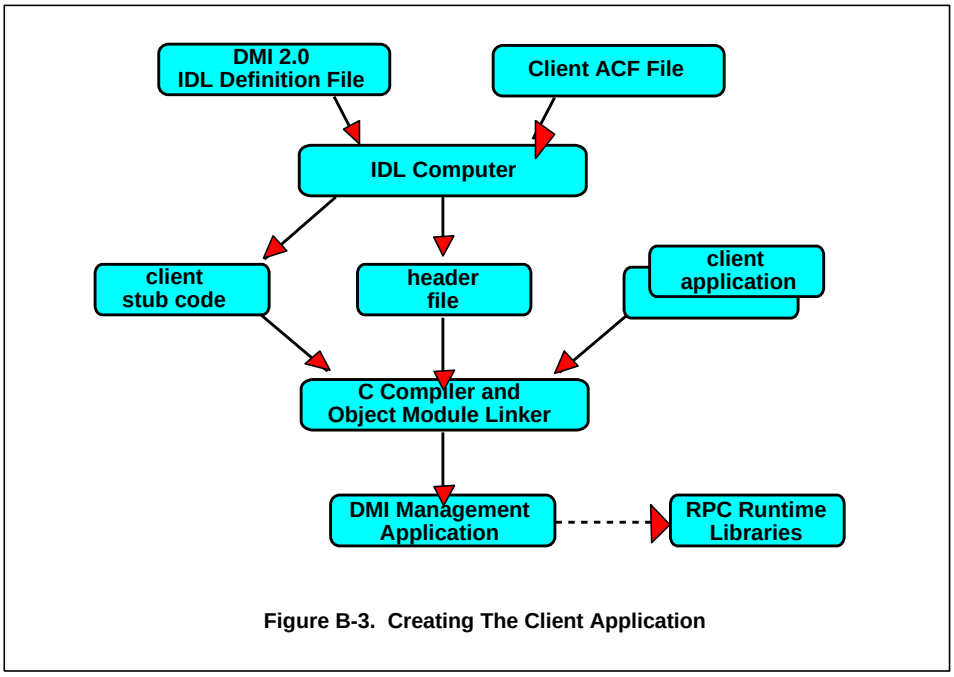


Figure B-3. Creating The Client Application

The first step in the development process is to create the client stub code and interface header file. As with the server case, this is done by compiling the DMTF-supplied IDL, along with client configuration information supplied in an ACF file. Next, you will build and compile your application code, then link everything together to create the RPC client application.

One of the first questions to answer when developing a management application is that of connecting, or *binding*, to the managed machine. The DMI 2.0 interface relies on standard RPC mechanisms to accomplish this binding.

To connect to a machine, a management application must supply:

- the machine's name or address,
- the protocol sequence (e.g.,, TCP/IP),
- and the Service Provider's process address (endpoint) on the managed machine.

A management application will typically specify the machine name and protocol sequence, and will most likely use a dynamically determined endpoint. This addressing data is used to construct a *binding handle;* binding handles are RPC-defined data structures that are used to manage the connection between RPC clients and servers.

Management applications that only talk to *one machine at a time* can construct an *implicit*, or global, binding handle. When used in this manner, the application is effectively saying that all remote procedure calls are directed toward a specific machine. When the application is done talking to that machine, it will free the binding. At this point, the application can construct a new binding handle for some other machine.

Management applications that *simultaneously* manage multiple machines will need to construct and maintain multiple binding handles: one per connection. In this usage model, the management application must explicitly supply a binding handle with each procedure call. This allows an application to direct procedure calls to different machines, while eliminating the need to create and free binding handles between procedure calls.

The IDL descriptions in this appendix ***do not*** include binding handles in the procedures' formal parameter lists. Instead, these API specifications concentrate on the DMI 2.0 interfaces themselves. If this is the case, then how can a management application select between explicit and implicit bindings? The answer can be found in the client's ACF file.

USING THE ACF FILE TO SPECIFY AN IMPLICIT BINDING HANDLE

If a client requires only one open connection at a time, it may choose to use an *implicit* binding handle. In this case, the contents of the ACF file would look like the following:

```
[ implicit_handle(handle_t dmi_server_binding_handle) ]
interface dmi_server
{
}
```

When this ACF file is supplied to the IDL compiler, the resulting header file will contain function prototypes that look exactly like those described in the IDL:

```
DmiErrorStatus_t
DmiRegister (DmiHandle_t* handle);
```

To use this implicit handle in your application, you first need to establish a binding to a remote machine, then perform the DMI 2.0 function calls, then unbind from the remote machine. Sample code for these actions might look something like the following example. The thing to note here is that we call the DMI 2.0 functions without explicitly passing a binding handle. A handle *does* exist, but it is stored within, and used by, the RPC stub code generated by the IDL compiler.

```
unsigned char*    string_binding;
unsigned32        status;

// The rpc_string_binding_compose function builds a string binding
// that can be used to bind an RPC client to a server.  There are
// other methods for binding to a remote machine; this is just the
// easiest to show.

rpc_string_binding_compose (NULL, "ncacn_ip_tcp", "your.machine.com",
                            NULL, NULL, &string_binding, &status);
CHECK_STATUS (status, ...);

// The rpc_binding_from_string_binding is where we actually bind
// the management application to the remote machine.  Note that
// we are passing the address of the dmi_server_binding_handle,
// which is the name declared in the ACF file.

rpc_binding_from_string_binding (string_binding,
                                 &dmi_server_binding_handle,
                                 &status);
CHECK_STATUS (status, ...);

// The rpc_string_free function is used to free the string storage
// allocated by the rpc_string_binding_compose function.

rpc_string_free (&string_binding, &status);
CHECK_STATUS (status, ...);

// Now we can perform any DMI 2.0 commands by simply calling
// the functions as if they were local procedure calls:

if (! statusOkay ((status = DmiRegister (&handle)))) {
    printf ("DmiRegister = %d\n", status);
    RAISE (status);
}

...

if (! statusOkay ((status = DmiUnregister (handle))))
```

```
    printf ("DmiUnregister = %d\n", status);

// Now we are done with our DMI 2.0 commands, so it's time
// to free up the binding so we can connect to someone else.

rpc_binding_free (&dmi_server_binding_handle, &status);
CHECK_STATUS (status);
```

USING THE ACF FILE TO SPECIFY AN EXPLICIT BINDING HANDLE

If you are building a client that requires simultaneous connections to different machines, you must use *explicit* binding handles. Explicit binding handles are stored and maintained in your application code; the RPC stub knows nothing about these handles. To use explicit binding handles, the contents of the ACF file would look like the following:

```
[ explicit_handle ]
interface dmi_server
{
}
```

When this ACF file is supplied to the IDL compiler, the resulting header file will contain function prototypes that contain an extra parameter in the formal parameter list. Note that *all* function prototypes will have this extra, binding handle parameter at the beginning of their parameter list. From this example, we can begin to see how the DMTF can define and publish an interface specification (the IDL), yet leave room for varying client implementations.

```
DmiErrorStatus_t
DmiRegister (handle_t        IDL_handle,
             DmiHandle_t*    handle);
```

To use this explicit handle in your application, you first need to establish bindings to the remote machines of interest, then perform the DMI 2.0 function calls, then unbind from the remote machines. Sample code for these actions might look something like the following example. The thing to note here is that we call the DMI 2.0 functions with an explicit binding handle, and that commands are interleaved from one machine to another.

```
rpc_binding_handle_t     binding_handle_1;
rpc_binding_handle_t     binding_handle_2;
unsigned char*           string_binding;
unsigned32               status;

// Bind the client to your.machine.com using TCP/IP.  This is
// identical to the implicit handle case, except that we are
// specifying that the binding information be stored in
// binding_handle_1.

rpc_string_binding_compose (NULL, "ncacn_ip_tcp", "your.machine.com",
                            NULL, NULL, &string_binding, &status);
CHECK_STATUS (status, ...);

rpc_binding_from_string_binding (string_binding, &binding_handle_1,
                                 &status);
CHECK_STATUS (status, ...);

rpc_string_free (&string_binding, &status);
CHECK_STATUS (status, ...);

// Bind the client to my.machine.com using TCP/IP. This is
// identical to the implicit handle case, except that we are
// specifying that the binding information be stored in
// binding_handle_2.

rpc_string_binding_compose (NULL, "ncacn_ip_tcp", "my.machine.com",
                            NULL, NULL, &string_binding, &status);
CHECK_STATUS (status, ...);

rpc_binding_from_string_binding (string_binding, &binding_handle_2,
                                 &status);
CHECK_STATUS (status, ...);

rpc_string_free (&string_binding, &status);
CHECK_STATUS (status, ...);

// Now we can perform DMI 2.0 commands to different machines by
```

```
// calling the procedures with different binding handles.

if (! statusOkay ((status = DmiRegister (binding_handle_1, &handle)))) {
    printf ("DmiRegister = %d\n", status);
    RAISE (status);
}

if (! statusOkay ((status = DmiRegister (binding_handle_2, &handle)))) {
    printf ("DmiRegister = %d\n", status);
    RAISE (status);
}

...

if (! statusOkay ((status = DmiRegister (binding_handle_2, &handle)))) {
    printf ("DmiRegister = %d\n", status);
    RAISE (status);
}

if (! statusOkay ((status = DmiRegister (binding_handle_1, &handle)))) {
    printf ("DmiRegister = %d\n", status);
    RAISE (status);
}

// Now we are done with our DMI 2.0 commands, so it's time
// to free up the bindings and leave.

rpc_binding_free (&binding_handle_1, &status);
CHECK_STATUS (status);

rpc_binding_free (&binding_handle_2, &status);
CHECK_STATUS (status);
```

USING THE ACF FILE TO CONTROL EXCEPTION HANDLING

We've seen how the ACF file can be used to select between implicit and explicit binding handles. There are several other client customizations that can be performed via the ACF file. Most notably, you can control whether or not your application receives exceptions from the RPC runtime system.

In the RPC environment, faults and communication errors are raised as exceptions to the RPC client. For example, if the client or server stub is unable to allocate sufficient memory for a procedure call, the client application may see an `rpc_x_no_memory` exception. Likewise, if there are communication errors, the client will see some communication-related exceptions, such as `rpc_x_comm_failure.` To handle these exceptions, a client will typically contain code with a TRY and CATCH block:

```
TRY {

    if (! statusOkay ((status = DmiUnregister (handle))))
        fprintf (efp, "DmiUnregister = %d\n", status);

} CATCH_ALL {

    // Put recovery code here

} ENDTRY;
```

If you don't want to use the RPC exception model, you can use the ACF file to change the behavior of the RPC stubs. To avoid exceptions entirely, specify an extra `status` parameter in the formal parameter list for all DMI 2.0 functions. The ACF syntax to perform this looks like the following:

```
[ implicit_handle(handle_t dmi_server_binding_handle) ]
interface dmi_server
{
    DmiErrorStatus_t
    DmiRegister ( DmiHandle_t*  handle,
                  [ comm_status, fault_status] status );
}
```

Here we have specified that both communication and fault exceptions for the DmiRegister function be reported in the `status` variable. It is possible to have some functions that raise exceptions, while others trap exceptions in a status variable. In practice, an application developer will probably pick one mechanism or another and stick with it for all

functions. With the above declaration, the IDL compiler will generate function prototypes that look like the following:

```
DmiErrorStatus_t
DmiRegister(
    DmiHandle_t*       handle,
    error_status_t*    status );
```

After each function call, the client application must check the status variable to see if any exceptions were trapped by the RPC stub.

COMMON DATA STRUCTURES (COMMON.IDL)

```
/*M*
//
//  RCS:
//      $Workfile:  common.idl $
//      $Revision:  2.0        $
//      $Modtime:   3/27/96    $
//      $Author:    DMTF       $
//
//  Purpose:
//
//      Describe data structures and types for the DMTF's Management
//      Interface in an IDL that is suitable for building remote
//      management using the DCE-RPC client/server model.  This
//      file is included in the client.idl and server.idl files.
//
//  Contents:
//
//      The following information is described in version 2.0
//      of the Desktop Management Interface Specification.
//
//  Enumerated Types:
//
//      DmiSetMode                  Define set operations
//      DmiRequestMode              Define sequential access modes
//      DmiStorageType              Define the storage type for an attribute
//      DmiAccessMode               Define the access modes for an attribute
//      DmiDataType                 Define the data types referenced by DmiDataUnion
//      DmiFileType                 Define the DMI mapping file types
//
//  Data Structures:
//
//      DmiTimestamp                Describes the DMI timestamp structure
//      DmiString                   Describes the DMI string representation
//      DmiOctetString              Describes the DMI octet string representation
//      DmiDataUnion                Discriminated union of DMI data types
//      DmiEnumInfo                 Associates an integer value with descriptive text
//      DmiAttributeInfo            Holds information about an attribute
//      DmiAttributeData            Describes an attribute id, type, and value
//      DmiGroupInfo                Holds information about a group
//      DmiComponentInfo            Holds information about a component
//      DmiFileDataInfo             Holds the schema file information: type and data
//      DmiClassNameInfo            Holds a group's id and class string
//      DmiRowRequest               Identifies { component, group, row, ids } to get
//      DmiRowData                  Identifies { component, group, row, values } to set
//
//      DmiAttributeIds             Describes a conformant array of DmiId
//      DmiAttributeValues          Describes a conformant array of
DmiAttributeData
//      DmiEnumList                 Describes a conformant array of DmiEnumInfo
//      DmiAttributeList            Describes a conformant array of DmiAttributeInfo
//      DmiGroupList                Describes a conformant array of DmiGroupInfo
//      DmiComponentList            Describes a conformant array of DmiComponentInfo
//      DmiFileDataList             Describes a conformant array of DmiFileDataInfo
//      DmiClassNameList            Describes a conformant array of DmiClassNameInfo
//      DmiStringList               Describes a conformant array of DmiString
//      DmiFileTypeList             Describes a conformant array of DmiFileType
//      DmiMultiRowRequest          Describes a conformant array of DmiRowRequest
//      DmiMultiRowData             Describes a conformant array of DmiRowData
*M*/

# ifndef DMI_API
# define DMI_API
# endif
```

```
/*********************************************************************
 * DmiSetMode
 *********************************************************************/

/*D*
//  Name:       DmiSetMode
//  Purpose:    Define set operations
//  Context:    DmiSetAttributes()
//  Fields:
//      DMI_SET        Set data values
//      DMI_RESERVE    Reserve resources for a set operation
//      DMI_RELEASE    Release previously reserved resources
*D*/

typedef enum {
    DMI_SET,
    DMI_RESERVE,
    DMI_RELEASE
} DmiSetMode_t;


/*********************************************************************
 * DmiRequestMode
 *********************************************************************/

/*D*
//  Name:       DmiRequestMode
//  Purpose:    Define sequential access modes
//  Context:    Field in DmiRowRequest,
//  Context:    DmiListComponents(), DmiListComponentsByClass(),
//  Context:    DmiListGroups(), DmiListAttributes(),
//  Fields:
//      DMI_UNIQUE    Access the specified item (or table row)
//      DMI_FIRST     Access the first item
//      DMI_NEXT      Access the next item
*D*/

typedef enum {
    DMI_UNIQUE,
    DMI_FIRST,
    DMI_NEXT
} DmiRequestMode_t;


/*********************************************************************
 * DmiStorageType
 *********************************************************************/

/*D*
//  Name:       DmiStorageType
//  Purpose:    Define the storage type for an attribute
//  Context:    Field in DmiAttributeInfo
//  Fields:
//      MIF_COMMON      Value is from a small set of possibilities
//      MIF_SPECIFIC    Value is from a large set of possibilities
*D*/

typedef enum {
    MIF_COMMON,
    MIF_SPECIFIC
} DmiStorageType_t;


/*********************************************************************
 * DmiAccessMode
 *********************************************************************/

/*D*
//  Name:       DmiAccessMode
//  Purpose:    Define the access modes for an attribute
//  Context:    Field in DmiAttributeInfo
//  Fields:
//      MIF_UNKNOWN        Unknown access mode
//      MIF_READ_ONLY      Read access only
```

```
//      MIF_READ_WRITE     Readable and writable
//      MIF_WRITE_ONLY     Write access only
//      MIF_UNSUPPORTED    Attribute is not supported
*D*/

typedef enum {
    MIF_UNKNOWN_ACCESS,
    MIF_READ_ONLY,
    MIF_READ_WRITE,
    MIF_WRITE_ONLY,
    MIF_UNSUPPORTED
} DmiAccessMode_t;


/*********************************************************************
 * DmiDataType
 *********************************************************************/

/*D*
//  Name:       DmiDataType
//  Purpose:    Define the data types referenced by DmiDataUnion
//  Context:
//  Fields:
//      MIF_DATATYPE_0       RESERVED
//      MIF_COUNTER          32-bit unsigned integer that never decreases
//      MIF_COUNTER64        64-bit unsigned integer that never decreases
//      MIF_GAUGE            32-bit unsigned integer may increase or decrease
//      MIF_DATATYPE_4       RESERVED
//      MIF_INTEGER          32-bit signed integer; no semantics known
//      MIF_INTEGER64        64-bit signed integer; no semantics known
//      MIF_OCTETSTRING      String of n octets, not necessarily displayable
//      MIF_DISPLAYSTRING    Displayable string of n octets
//      MIF_DATATYPE_9       RESERVED
//      MIF_DATATYPE_10      RESERVED
//      MIF_DATE             28-octet displayable string
(yyyymmddHHMMSS.uuuuuu+ooo)
*D*/

typedef enum {
    MIF_DATATYPE_0,
    MIF_COUNTER,
    MIF_COUNTER64,
    MIF_GAUGE,
    MIF_DATATYPE_4,
    MIF_INTEGER,
    MIF_INTEGER64,
    MIF_OCTETSTRING,
    MIF_DISPLAYSTRING,
    MIF_DATATYPE_9,
    MIF_DATATYPE_10,
    MIF_DATE
} DmiDataType_t;

/*
 * Aliases for the standard data types
 */

# define MIF_INT     MIF_INTEGER
# define MIF_INT64   MIF_INTEGER64
# define MIF_STRING  MIF_DISPLAYSTRING


/*********************************************************************
 * DmiFileType
 *********************************************************************/

/*D*
//  Name:       DmiFileType
//  Purpose:    Define the DMI mapping file types
//  Context:    Field in DmiFileDataInfo
//  Fields:
//      DMI_FILETYPE_0            RESERVED
//      DMI_FILETYPE_1            RESERVED
//      DMI_MIF_FILE_NAME         File data is DMI MIF file name
//      DMI_MIF_FILE_DATA         File data is DMI MIF data
```

```
//      SNMP_MAPPING_FILE_NAME    File data is SNMP MAPPING file name
//      SNMP_MAPPING_FILE_DATA    File data is SNMP MAPPING data
//      DMI_GROUP_FILE_NAME       File data is DMI GROUP MIF file name
//      DMI_GROUP_FILE_DATA       File data is DMI GROUP MIF data
//      VENDOR_FORMAT_FILE_NAME   File data is Vendor specific file name
//      VENDOR_FORMAT_FILE_DATA   File data is Vendor specific data
*D*/

typedef enum {
    DMI_FILETYPE_0,
    DMI_FILETYPE_1,
    DMI_MIF_FILE_NAME,
    DMI_MIF_FILE_DATA,
    SNMP_MAPPING_FILE_NAME,
    SNMP_MAPPING_FILE_DATA,
    DMI_GROUP_FILE_NAME,
    DMI_GROUP_FILE_DATA,
    VENDOR_FORMAT_FILE_NAME,
    VENDOR_FORMAT_FILE_DATA
} DmiFileType_t;


/*********************************************************************
 * DMI Data Types
 *********************************************************************/

typedef unsigned long   DmiId_t;
typedef unsigned long   DmiHandle_t;
typedef unsigned long   DmiCounter_t;
typedef unsigned long   DmiErrorStatus_t;
typedef unsigned hyper  DmiCounter64_t;
typedef unsigned long   DmiGauge_t;
typedef unsigned long   DmiUnsigned_t;
typedef long            DmiInteger_t;
typedef hyper           DmiInteger64_t;
typedef boolean         DmiBoolean_t;


/*********************************************************************
 * DmiTimestamp
 *********************************************************************/

/*D*
//  Name:       DmiTimestamp
//  Purpose:    Describes the DMI timestamp structure
//  Context:    Field in DmiDataUnion
//  Fields:
//      year            The year ('1996')
//      month           The month ('1'..'12')
//      yay             The day of the month ('1'..'23')
//      hour            The hour ('0'..'23')
//      minutes         The minutes ('0'..'59')
//      seconds         The seconds ('0'..'60'); includes leap seconds
//      dot             A dot ('.')
//      microSeconds    Microseconds ('0'..'999999')
//      plusOrMinus     '+' for east, or '-' west of UTC
//      utcOffset       Minutes ('0'..'720') from UTC
//      padding         Unused padding for 4-byte alignment
*D*/

typedef struct DmiTimestamp {
    char  year          [4];
    char  month         [2];
    char  day           [2];
    char  hour          [2];
    char  minutes       [2];
    char  seconds       [2];
    char  dot;
    char  microSeconds  [6];
    char  plusOrMinus;
    char  utcOffset     [3];
    char  padding       [3];
} DmiTimestamp_t;
```

```
/*********************************************************************
 * DmiString
 *********************************************************************/

/*D*
//  Name:       DmiString
//  Purpose:    Describes the DMI string representation
//  Context:    Field in DmiDataUnion
//  Fields:
//      size    Number of octets in the string body
//      body    String contents
//
//  Notes:      For displaystrings, the string is null terminated,
//              and the null character is included in the size.
*D*/

typedef struct DmiString {
    DmiUnsigned_t  size;
    [size_is (size)] char*  body;
} DmiString_t;


typedef DmiString_t* DmiStringPtr_t;


/*********************************************************************
 * DmiOctetString
 *********************************************************************/

/*D*
//  Name:       DmiOctetString
//  Purpose:    Describes the DMI octet string representation
//  Context:    Field in DmiDataUnion
//  Fields:
//      size    Number of octets in the string body
//      body    String contents
*D*/

typedef struct DmiOctetString {
    DmiUnsigned_t  size;
    [size_is (size)] char*  body;
} DmiOctetString_t;


/*********************************************************************
 * DmiDataUnion
 *********************************************************************/

/*D*
//  Name:       DmiDataUnion
//  Purpose:    Discriminated union of DMI data types
//  Context:    Field in DmiAttributeData
//  Fields:
//      type    Discriminator for the union
//      value   Union of DMI attribute data types
*D*/

typedef union DmiDataUnion
    switch (DmiDataType_t type) value {
    case MIF_COUNTER:       DmiCounter_t       counter;
    case MIF_COUNTER64:     DmiCounter64_t     counter64;
    case MIF_GAUGE:         DmiGauge_t         gauge;
    case MIF_INTEGER:       DmiInteger_t       integer;
    case MIF_INTEGER64:     DmiInteger64_t     integer64;
    case MIF_OCTETSTRING:   DmiOctetString_t*  octetstring;
    case MIF_DISPLAYSTRING: DmiString_t*       string;
    case MIF_DATE:          DmiTimestamp_t*    date;
} DmiDataUnion_t;
```

```
/*********************************************************************
 * DmiEnumInfo
 *********************************************************************/

/*D*
//  Name:       DmiEnumInfo
//  Purpose:    Associates an integer value with descriptive text
//  Context:    Element in DmiEnumList
//  Fields:
//      name    Enumeration name
//      value   Enumeration value
*D*/

typedef struct DmiEnumInfo {
    DmiString_t*    name;
    DmiInteger_t    value;
} DmiEnumInfo_t;


/*********************************************************************
 * DmiAttributeInfo
 *********************************************************************/

/*D*
//  Name:       DmiAttributeInfo
//  Purpose:    Holds information about an attribute
//  Context:    Element in DmiAttributeList
//  Fields:
//      id              Attribute ID
//      name            Attribute name string
//      pragma          Attribute pragma string             [optional]
//      description     Attribute description string       [optional]
//      storage         Common or specific storage
//      access          Readonly, read-write, etc
//      type            Counter, integer, etc
//      maxSize         Maximum length of the attribute
//      enumList        EnumList for enumerated types       [optional]
*D*/

typedef struct DmiAttributeInfo {
    DmiId_t                 id;
    DmiString_t*            name;
    DmiString_t*            pragma;
    DmiString_t*            description;
    DmiStorageType_t        storage;
    DmiAccessMode_t         access;
    DmiDataType_t           type;
    DmiUnsigned_t           maxSize;
    struct DmiEnumList*     enumList;
} DmiAttributeInfo_t;


/*********************************************************************
 * DmiAttributeData
 *********************************************************************/

/*D*
//  Name:       DmiAttributeData
//  Purpose:    Describes an attribute id, type, and value
//  Context:    Element in DmiAttributeValues
//  Fields:
//      id      Attribute ID
//      data    Attribute type and value
*D*/

typedef struct DmiAttributeData {
    DmiId_t           id;
    DmiDataUnion_t    data;
} DmiAttributeData_t;
```

```
/*********************************************************************
 * DmiGroupInfo
 *********************************************************************/

/*D*
//  Name:       DmiGroupInfo
//  Purpose:    Holds information about a group
//  Context:    Element in DmiGroupList
//  Fields:
//      id              Group ID
//      name            Group name string
//      pragma          Group pragma string                 [optional]
//      className       Group class name string
//      description     Group description string            [optional]
//      keyList         Attribute IDs for table row keys    [optional]
*D*/

typedef struct DmiGroupInfo {
    DmiId_t                     id;
    DmiString_t*                name;
    DmiString_t*                pragma;
    DmiString_t*                className;
    DmiString_t*                description;
    struct DmiAttributeIds*     keyList;
} DmiGroupInfo_t;


/*********************************************************************
 * DmiComponentInfo
 *********************************************************************/

/*D*
//  Name:       DmiComponentInfo
//  Purpose:    Holds information about a component
//  Context:    Element in DmiComponentList
//  Fields:
//      id              Component ID
//      name            Component name string
//      pragma          Component pragma string         [optional]
//      description     Component description string    [optional]
//      exactMatch
//         idl_true  = Exact match
//         idl_false = Possible match
*D*/

typedef struct DmiComponentInfo {
    DmiId_t         id;
    DmiString_t*    name;
    DmiString_t*    pragma;
    DmiString_t*    description;
    DmiBoolean_t    exactMatch;
} DmiComponentInfo_t;


/*********************************************************************
 * DmiFileDataInfo
 *********************************************************************/

/*D*
//  Name:       DmiFileDataInfo
//  Purpose:    Holds the schema file information: type and data
//  Context:    Element in DmiFileDataList
//  Fields:
//      fileType    MIF file, SNMP mapping file, etc
//      fileData    The file info (name -or- contents)
*D*/

typedef struct DmiFileDataInfo {
    DmiFileType_t        fileType;
    DmiOctetString_t*    fileData;
} DmiFileDataInfo_t;
```

```
/*********************************************************************
 * DmiClassNameInfo
 *********************************************************************/

/*D*
//  Name:       DmiClassNameInfo
//  Purpose:    Holds a group's id and class string
//  Context:    Element in DmiClassNameList
//  Fields:
//      id          Group ID
//      className   Group class name string
*D*/

typedef struct DmiClassNameInfo {
    DmiId_t       id;
    DmiString_t*  className;
} DmiClassNameInfo_t;


/*********************************************************************
 * DmiRowRequest
 *********************************************************************/

/*D*
//  Name:       DmiRowRequest
//  Purpose:    Identifies { component, group, row, ids } to get
//  Context:    Element in DmiMultiRowRequest
//  Fields:
//      compId         Component ID
//      groupId        Group ID
//      requestMode    Get from specified row, first row, or next row
//      keyList        Array of values for key attributes
//      ids            Array of IDs for data attributes
*D*/

typedef struct DmiRowRequest {
    DmiId_t                      compId;
    DmiId_t                      groupId;
    DmiRequestMode_t             requestMode;
    struct DmiAttributeValues*   keyList;
    struct DmiAttributeIds*      ids;
} DmiRowRequest_t;


/*********************************************************************
 * DmiRowData
 *********************************************************************/

/*D*
//  Name:       DmiRowData
//  Purpose:    Identifies { component, group, row, values } to set
//  Context:    Element in DmiMultiRowData
//  Fields:
//      compId       Component ID
//      groupId      Group ID
//      className    Group class name for events, or 0    [optional]
//      keyList      Array of values for key attributes
//      values       Array of values for data attributes
//
//  Notes:      This structure is used for setting attributes, getting
//              attributes, and for providing indication data.  The
//              className string is only required when returning
//              indication data.  For other uses, the field can be 0.
*D*/

typedef struct DmiRowData {
    DmiId_t                      compId;
    DmiId_t                      groupId;
    DmiString_t*                 className;
    struct DmiAttributeValues*   keyList;
    struct DmiAttributeValues*   values;
} DmiRowData_t;
```

```
/*********************************************************************
 * DmiAttributeIds
 *********************************************************************/

/*D*
//  Name:       DmiAttributeIds
//  Purpose:    Describes a conformant array of DmiId
//  Context:    Field in DmiRowRequest
//  Fields:
//      size    Array elements
//      list    Array data
*D*/

typedef struct DmiAttributeIds {
    DmiUnsigned_t  size;
    [size_is (size)] DmiId_t*  list;
} DmiAttributeIds_t;


/*********************************************************************
 * DmiAttributeValues
 *********************************************************************/

/*D*
//  Name:       DmiAttributeValues
//  Purpose:    Describes a conformant array of DmiAttributeData
//  Context:    Field in DmiRowRequest, DmiRowData
//  Fields:
//      size    Array elements
//      list    Array data
*D*/

typedef struct DmiAttributeValues {
    DmiUnsigned_t  size;
    [size_is (size)] DmiAttributeData_t*  list;
} DmiAttributeValues_t;


/*********************************************************************
 * DmiEnumList
 *********************************************************************/

/*D*
//  Name:       DmiEnumList
//  Purpose:    Describes a conformant array of DmiEnumInfo
//  Context:    DmiEnumAttributes()
//  Fields:
//      size    Array elements
//      list    Array data
*D*/

typedef struct DmiEnumList {
    DmiUnsigned_t  size;
    [size_is (size)] DmiEnumInfo_t*  list;
} DmiEnumList_t;


/*********************************************************************
 * DmiAttributeList
 *********************************************************************/

/*D*
//  Name:       DmiAttributeList
//  Purpose:    Describes a conformant array of DmiAttributeInfo
//  Context:    DmiListAttributes()
//  Fields:
//      size    Array elements
//      list    Array data
*D*/

typedef struct DmiAttributeList {
    DmiUnsigned_t  size;
    [size_is (size)] DmiAttributeInfo_t*  list;
} DmiAttributeList_t;
```

```
/*********************************************************************
 * DmiGroupList
 *********************************************************************/

/*D*
//  Name:       DmiGroupList
//  Purpose:    Describes a conformant array of DmiGroupInfo
//  Context:    DmiListGroups()
//  Fields:
//      size    Array elements
//      list    Array data
*D*/

typedef struct DmiGroupList {
    DmiUnsigned_t  size;
    [size_is (size)] DmiGroupInfo_t*  list;
} DmiGroupList_t;


/*********************************************************************
 * DmiComponent
 *********************************************************************/

/*D*
//  Name:       DmiComponentList
//  Purpose:    Describes a conformant array of DmiComponentInfo
//  Context:    DmiListComponents(), DmiListComponentsByClass()
//  Fields:
//      size    Array elements
//      list    Array data
*D*/

typedef struct DmiComponentList {
    DmiUnsigned_t  size;
    [size_is (size)] DmiComponentInfo_t*  list;
} DmiComponentList_t;


/*********************************************************************
 * DmiFileDataList
 *********************************************************************/

/*D*
//  Name:       DmiFileDataList
//  Purpose:    Describes a conformant array of DmiFileDataInfo
//  Context:    DmiAddComponent(), DmiAddLanguage(), DmiAddGroup()
//  Fields:
//      size    Array elements
//      list    Array data
*D*/

typedef struct DmiFileDataList {
    DmiUnsigned_t  size;
    [size_is (size)] DmiFileDataInfo_t*  list;
} DmiFileDataList_t;


/*********************************************************************
 * DmiClassNameList
 *********************************************************************/

/*D*
//  Name:       DmiClassNameList
//  Purpose:    Describes a conformant array of DmiClassNameInfo
//  Context:    DmiListClassNames()
//  Fields:
//      size    Array elements
//      list    Array data
*D*/

typedef struct DmiClassNameList {
    DmiUnsigned_t  size;
    [size_is (size)] DmiClassNameInfo_t*  list;
} DmiClassNameList_t;
```

```
/*********************************************************************
 * DmiStringList
 *********************************************************************/

/*D*
//  Name:       DmiStringList
//  Purpose:    Describes a conformant array of DmiStrings
//  Context:    DmiListLanguages()
//  Fields:
//      size    Array elements
//      list    Array data
*D*/

typedef struct DmiStringList {
    DmiUnsigned_t  size;
    [size_is (size)] DmiStringPtr_t*  list;
} DmiStringList_t;


/*********************************************************************
 * DmiFileTypeList
 *********************************************************************/

/*D*
//  Name:       DmiFileTypeList
//  Purpose:    Describes a conformant array of DmiFileType entries
//  Context:    DmiGetVersion()
//  Fields:
//      size    Array elements
//      list    Array data
*D*/

typedef struct DmiFileTypeList {
    DmiUnsigned_t  size;
    [size_is (size)] DmiFileType_t*  list;
} DmiFileTypeList_t;


/*********************************************************************
 * DmiMultiRowRequest
 *********************************************************************/

/*D*
//  Name:       DmiMultiRowRequest
//  Purpose:    Describes a conformant array of DmiRowRequest
//  Context:    DmiGetAttributes()
//  Fields:
//      size    Array elements
//      list    Array data
*D*/

typedef struct DmiMultiRowRequest {
    DmiUnsigned_t  size;
    [size_is (size)] DmiRowRequest_t*  list;
} DmiMultiRowRequest_t;


/*********************************************************************
 * DmiMultiRowData
 *********************************************************************/

/*D*
//  Name:       DmiMultiRowData
//  Purpose:    Describes a conformant array of DmiRowData
//  Context:    DmiGetAttributes(), DmiSetAttributes()
//  Fields:
//      size    Array elements
//      list    Array data
*D*/

typedef struct DmiMultiRowData {
    DmiUnsigned_t  size;
    [size_is (size)] DmiRowData_t*  list;
} DmiMultiRowData_t;
```

MANAGEMENT INTERFACE (SERVER.IDL)

```
/*M*
//
//  RCS:
//      $Workfile:  server.idl $
//      $Revision:  2.0        $
//      $Modtime:   3/27/96    $
//      $Author:    DMTF       $
//
//  Purpose:
//
//      Describe the DMTF's Management Interface in an IDL that is
//      suitable for building remote management using the DCE-RPC
//      client/server model.  This file, along with server.acf,
//      is compiled with the IDL compiler to produce the following
//      files:
//
//                server.h          C-style interface header file
//                server_c.c        Stub code for the rmi client
//                server_s.c        Stub code for the rmi server
//
//  Contents:
//
//      The following information is described in version 2.0
//      of the Desktop Management Interface Specification.
//
//  Initialization:
//
//      DmiRegister                 Register a session with a remote system
//      DmiUnregister               Unregister a previously registered session
//      DmiGetVersion               Get DMI Service Provider version information
//      DmiGetConfig                Get session configuration parameters
//      DmiSetConfig                Set session configuration parameters
//
//  Discovery:
//
//      DmiListComponents           List component properties
//      DmiListComponentsByClass    List components matching certain criteria
//      DmiListLanguages            List a component's language strings
//      DmiListClassNames           List a component's class names and group ids
//      DmiListGroups               List group properties
//      DmiListAttributes           List attribute properties
//
//  Operation:
//
//      DmiAddRow                   Add a new row to a table
//      DmiDeleteRow                Delete a row from a table
//      DmiGetAttribute             Get a single attribute value
//      DmiSetAttribute             Set a single attribute value
//      DmiGetMultiple              Get a collection of attribute values
//      DmiSetMultiple              Set a collection of attribute values
//
//  Database Administration:
//
//      DmiAddComponent             Add a new component to the DMI database
//      DmiAddLanguage              Add a new language mapping for a component
//      DmiAddGroup                 Add a new group to a component
//      DmiDeleteComponent          Delete a component from the DMI database
//      DmiDeleteLanguage           Delete a language mapping for a component
//      DmiDeleteGroup              Delete a group from a component
*M*/

[
    uuid(892b2b90-1532-11cf-9a39-00aa0034b922),
    version(2.0),
    pointer_default(ptr)
]
    interface dmi_server
{

# include "common.idl"
```

```
/*************************************************************************
 * DmiRegister
 *************************************************************************/

/*F*
//  Name:       DmiRegister
//  Purpose:    Register a session with a remote system
//  Context:    Initialization
//  Returns:
//  Parameters:
//      handle    On completion, an open session handle
//
//  Notes:      The client provides the address of the handle
//              parameter and the server fills it in.  All commands
//              except DmiRegister() require a valid handle, so
//              this must be the first command sent to the DMI server.
*F*/

DmiErrorStatus_t  DMI_API
DmiRegister (
    [out]  DmiHandle_t*  handle );


/*************************************************************************
 * DmiUnregister
 *************************************************************************/

/*F*
//  Name:       DmiUnregister
//  Purpose:    Unregister a previously registered session
//  Context:    Initialization
//  Returns:
//  Parameters:
//      handle    An open session handle to be closed
*F*/

DmiErrorStatus_t  DMI_API
DmiUnregister (
    [in]  DmiHandle_t  handle );


/*************************************************************************
 * DmiGetVersion
 *************************************************************************/

/*F*
//  Name:       DmiGetVersion
//  Purpose:    Get DMI Service Provider version information
//  Context:    Initialization
//  Returns:
//  Parameters:
//      handle           An open session handle
//      dmiSpecLevel     The DMI Specification version
//      description      The OS-specific Service Provider version
//      fileTypes        Supported file types for schema description
//
//  Notes:      1. The client must free the dmiSpecLevel string
//              2. The client must free the description string
*F*/

DmiErrorStatus_t  DMI_API
DmiGetVersion (
    [in]   DmiHandle_t          handle,
    [out]  DmiString_t**        dmiSpecLevel,
    [out]  DmiString_t**        description,
    [out]  DmiFileTypeList_t**  fileTypes );
```

```
/*********************************************************************
 * DmiGetConfig
 *********************************************************************/

/*F*
//  Name:       DmiGetConfig
//  Purpose:    Get session configuration parameters
//  Context:    Initialization
//  Returns:
//  Parameters:
//      handle      An open session handle
//      language    language-code|territory-code|encoding
//
//  Notes:      The client must free the language string
*F*/

DmiErrorStatus_t  DMI_API
DmiGetConfig (
    [in]   DmiHandle_t    handle,
    [out]  DmiString_t**  language );


/*********************************************************************
 * DmiSetConfig
 *********************************************************************/

/*F*
//  Name:       DmiSetConfig
//  Purpose:    Set session configuration parameters
//  Context:    Initialization
//  Returns:
//  Parameters:
//      handle      An open session handle
//      language    language-code|territory-code|encoding
*F*/

DmiErrorStatus_t  DMI_API
DmiSetConfig (
    [in]  DmiHandle_t   handle,
    [in]  DmiString_t*  language );


/*********************************************************************
 * Dmilistcomponents
 *********************************************************************/

/*F*
//  Name:       DmiListComponents
//  Purpose:    List component properties
//  Context:    Discovery
//  Returns:
//  Parameters:
//      handle            An open session handle
//      requestMode       Unique, first, or next component ?
//      maxCount          Maximum number to return, or 0 for all
//      getPragma         Get optional pragma string ?
//      getDescription    Get optional component description ?
//      compId            Component to start with (see requestMode)
//      reply             List of components
//
//  Notes:      The client must free the reply structure
*F*/

DmiErrorStatus_t  DMI_API
DmiListComponents (
    [in]   DmiHandle_t           handle,
    [in]   DmiRequestMode_t      requestMode,
    [in]   DmiUnsigned_t         maxCount,
    [in]   DmiBoolean_t          getPragma,
    [in]   DmiBoolean_t          getDescription,
    [in]   DmiId_t               compId,
    [out]  DmiComponentList_t**  reply );
```

```
/*****************************************************************
 * DmiListComponentsByClass
 *****************************************************************/

/*F*
//  Name:       DmiListComponentsByClass
//  Purpose:    List components matching certain criteria
//  Context:    Discovery
//  Returns:
//  Parameters:
//      handle            An open session handle
//      requestMode       Unique, first, or next component ?
//      maxCount          Maximum number to return, or 0 for all
//      getPragma         Get optional pragma string ?
//      getDescription    Get optional component description ?
//      compId            Component to start with (see requestMode)
//      className         Group class name string to match
//      keyList           Group row keys to match, or null
//      reply             List of components
//
//  Notes:      The client must free the reply structure
*F*/

DmiErrorStatus_t  DMI_API
DmiListComponentsByClass (
    [in]       DmiHandle_t            handle,
    [in]       DmiRequestMode_t       requestMode,
    [in]       DmiUnsigned_t          maxCount,
    [in]       DmiBoolean_t           getPragma,
    [in]       DmiBoolean_t           getDescription,
    [in]       DmiId_t                compId,
    [in]       DmiString_t*           className,
    [in, ptr]  DmiAttributeValues_t*  keyList,
    [out]      DmiComponentList_t**   reply );


/*****************************************************************
 * DmiListLanguages
 *****************************************************************/

/*F*
//  Name:       DmiListLanguages
//  Purpose:    List a component's language strings
//  Context:    Discovery
//  Returns:
//  Parameters:
//      handle         An open session handle
//      maxCount       Maximum number to return, or 0 for all
//      compId         Component to access
//      reply          List of language strings
//
//  Notes:      The client must free the reply structure
*F*/

DmiErrorStatus_t  DMI_API
DmiListLanguages (
    [in]   DmiHandle_t        handle,
    [in]   DmiUnsigned_t      maxCount,
    [in]   DmiId_t            compId,
    [out]  DmiStringList_t**  reply );


/*****************************************************************
 * DmiListClassNames
 *****************************************************************/

/*F*
//  Name:       DmiListClassNames
//  Purpose:    List a component's class names and group ids
//  Context:    Discovery
//  Returns:
//  Parameters:
//      handle         An open session handle
//      maxCount       Maximum number to return, or 0 for all
```

```
//      compId         Component to access
//      reply          List of class names and group ids
//
//  Notes:      The client must free the reply structure
*F*/

DmiErrorStatus_t  DMI_API
DmiListClassNames (
    [in]   DmiHandle_t           handle,
    [in]   DmiUnsigned_t         maxCount,
    [in]   DmiId_t               compId,
    [out]  DmiClassNameList_t**  reply );


/*********************************************************************
 * DmiListGroups
 *********************************************************************/

/*F*
//  Name:       DmiListGroups
//  Purpose:    List group properties
//  Context:    Discovery
//  Returns:
//  Parameters:
//      handle            An open session handle
//      requestMode       Unique, first, or next group ?
//      maxCount          Maximum number to return, or 0 for all
//      getPragma         Get optional pragma string ?
//      getDescription    Get optional group description ?
//      compId            Component to access
//      groupId           Group to start with (see requestMode)
//      reply             List of groups
//
//  Notes:      The client must free the reply structure
*F*/

DmiErrorStatus_t  DMI_API
DmiListGroups (
    [in]   DmiHandle_t        handle,
    [in]   DmiRequestMode_t   requestMode,
    [in]   DmiUnsigned_t      maxCount,
    [in]   DmiBoolean_t       getPragma,
    [in]   DmiBoolean_t       getDescription,
    [in]   DmiId_t            compId,
    [in]   DmiId_t            groupId,
    [out]  DmiGroupList_t**   reply );


/*********************************************************************
 * DmiListAttributes
 *********************************************************************/

/*F*
//  Name:       DmiListAttributes
//  Purpose:    List attribute properties
//  Context:    Discovery
//  Returns:
//  Parameters:
//      handle            An open session handle
//      requestMode       Unique, first, or next attribute ?
//      maxCount          Maximum number to return, or 0 for all
//      getPragma         Get optional pragma string ?
//      getDescription    Get optional attribute description ?
//      compId            Component to access
//      groupId           Group to access
//      attribId          Attribute to start with (see requestMode)
//      reply             List of attributes
//
//  Notes:      The client must free the reply structure
*F*/

DmiErrorStatus_t  DMI_API
DmiListAttributes (
    [in]   DmiHandle_t           handle,
    [in]   DmiRequestMode_t      requestMode,
```

```
    [in]   DmiUnsigned_t         maxCount,
    [in]   DmiBoolean_t          getPragma,
    [in]   DmiBoolean_t          getDescription,
    [in]   DmiId_t               compId,
    [in]   DmiId_t               groupId,
    [in]   DmiId_t               attribId,
    [out]  DmiAttributeList_t**  reply );


/*********************************************************************
 * DmiAddComponent
 *********************************************************************/

/*F*
//  Name:       DmiAddComponent
//  Purpose:    Add a new component to the DMI database
//  Context:    Database Administration
//  Returns:
//  Parameters:
//      handle      An open session handle
//      fileData    Schema description for the component
//      compId      On completion, the SP-allocated component id
//      errors      Installation error messages
*F*/

DmiErrorStatus_t  DMI_API
DmiAddComponent (
    [in]   DmiHandle_t         handle,
    [in]   DmiFileDataList_t*  fileData,
    [out]  DmiId_t*            compId,
    [out]  DmiStringList_t**   errors );


/*********************************************************************
 * DmiAddLanguage
 *********************************************************************/

/*F*
//  Name:       DmiAddLanguage
//  Purpose:    Add a new language mapping for a component
//  Context:    Database Administration
//  Returns:
//  Parameters:
//      handle      An open session handle
//      fileData    Language mapping file for the component
//      compId      Component to access
//      errors      Installation error messages
*F*/

DmiErrorStatus_t  DMI_API
DmiAddLanguage (
    [in]   DmiHandle_t         handle,
    [in]   DmiFileDataList_t*  fileData,
    [in]   DmiId_t             compId,
    [out]  DmiStringList_t**   errors );


/*********************************************************************
 * DmiAddGroup
 *********************************************************************/

/*F*
//  Name:       DmiAddGroup
//  Purpose:    Add a new group to a component
//  Context:    Database Administration
//  Returns:
//  Parameters:
//      handle      An open session handle
//      fileData    Schema description for the group
//      compId      Component to access
//      groupId     On completion, the SP-allocated group ID
//      errors      Installation error messages
*F*/
```

```
DmiErrorStatus_t  DMI_API
DmiAddGroup (
    [in]   DmiHandle_t         handle,
    [in]   DmiFileDataList_t*  fileData,
    [in]   DmiId_t             compId,
    [out]  DmiId_t*            groupId,
    [out]  DmiStringList_t**   errors );


/*************************************************************************
 * DmiDeleteComponent
 *************************************************************************/

/*F*
//  Name:       DmiDeleteComponent
//  Purpose:    Delete a component from the DMI database
//  Context:    Database Administration
//  Returns:
//  Parameters:
//      handle    An open session handle
//      compId    Component to delete
*F*/

DmiErrorStatus_t  DMI_API
DmiDeleteComponent (
    [in]  DmiHandle_t  handle,
    [in]  DmiId_t      compId );


/*************************************************************************
 * DmiDeleteLanguage
 *************************************************************************/

/*F*
//  Name:       DmiDeleteLanguage
//  Purpose:    Delete a language mapping for a component
//  Context:    Database Administration
//  Returns:
//  Parameters:
//      handle      An open session handle
//      language    language-code|territory-code|encoding
//      compId      Component to access
*F*/

DmiErrorStatus_t  DMI_API
DmiDeleteLanguage (
    [in]  DmiHandle_t   handle,
    [in]  DmiString_t*  language,
    [in]  DmiId_t       compId );


/*************************************************************************
 * DmiDeleteGroup
 *************************************************************************/

/*F*
//  Name:       DmiDeleteGroup
//  Purpose:    Delete a group from a component
//  Context:    Database Administration
//  Returns:
//  Parameters:
//      handle     An open session handle
//      compId     Component containing group
//      groupId    Group to delete
*F*/

DmiErrorStatus_t  DMI_API
DmiDeleteGroup (
    [in]  DmiHandle_t  handle,
    [in]  DmiId_t      compId,
    [in]  DmiId_t      groupId );
```

```
/*****************************************************************
 * DmiAddRow
 *****************************************************************/

/*F*
//  Name:       DmiAddRow
//  Purpose:    Add a new row to a table
//  Context:    Operation
//  Returns:
//  Parameters:
//      handle    An open session handle
//      rowData   Attribute values to set
*F*/

DmiErrorStatus_t  DMI_API
DmiAddRow (
    [in]  DmiHandle_t    handle,
    [in]  DmiRowData_t*  rowData );


/*****************************************************************
 * DmiDeleteRow
 *****************************************************************/

/*F*
//  Name:       DmiDeleteRow
//  Purpose:    Delete a row from a table
//  Context:    Operation
//  Returns:
//  Parameters:
//      handle     An open session handle
//      rowData    Row { component, group, key } to delete
*F*/

DmiErrorStatus_t  DMI_API
DmiDeleteRow (
    [in]  DmiHandle_t    handle,
    [in]  DmiRowData_t*  rowData );


/*****************************************************************
 * DmiGetAttribute
 *****************************************************************/

/*F*
//  Name:       DmiGetAttribute
//  Purpose:    Get a single attribute value
//  Context:    Operation
//  Returns:
//  Parameters:
//      handle      An open session handle
//      compId      Component to access
//      groupId     Group within component
//      attribId    Attribute within group
//      keyList     Keylist to specify a table row    [optional]
//      value       Attribute value returned
*F*/

DmiErrorStatus_t  DMI_API
DmiGetAttribute (
    [in]       DmiHandle_t            handle,
    [in]       DmiId_t                compId,
    [in]       DmiId_t                groupId,
    [in]       DmiId_t                attribId,
    [in, ptr]  DmiAttributeValues_t*  keyList,
    [out]      DmiDataUnion_t**       value );


/*****************************************************************
 * DmiSetAttribute
 *****************************************************************/

/*F*
//  Name:       DmiSetAttribute
//  Purpose:    Set a single attribute value
```

```
//  Context:    Operation
//  Returns:
//  Parameters:
//      handle      An open session handle
//      compId      Component to access
//      groupId     Group within component
//      attribId    Attribute within group
//      keyList     Keylist to specify a table row    [optional]
//      setMode     Set, reserve, or release ?
//      value       Attribute value to set
*F*/

DmiErrorStatus_t  DMI_API
DmiSetAttribute (
    [in]       DmiHandle_t            handle,
    [in]       DmiId_t                compId,
    [in]       DmiId_t                groupId,
    [in]       DmiId_t                attribId,
    [in, ptr]  DmiAttributeValues_t*  keyList,
    [in]       DmiSetMode_t           setMode,
    [in]       DmiDataUnion_t*        value );


/*********************************************************************
 * DmiGetMultiple
 *********************************************************************/

/*F*
//  Name:       DmiGetMultiple
//  Purpose:    Get a collection of attribute values
//  Context:    Operation
//  Returns:
//  Parameters:
//      handle     An open session handle
//      request    Attributes to get
//      rowData    Requested attribute values
//
//  Notes:      1. The request may be for a SINGLE row (size = 1)
//              2. An empty id list for a row means "get all attributes"
//              3. The client must free the rowData structure
*F*/

DmiErrorStatus_t  DMI_API
DmiGetMultiple (
    [in]   DmiHandle_t            handle,
    [in]   DmiMultiRowRequest_t*  request,
    [out]  DmiMultiRowData_t**    rowData );


/*********************************************************************
 * DmiSetMultiple
 *********************************************************************/

/*F*
//  Name:       DmiSetMultiple
//  Purpose:    Set a collection of attributes
//  Context:    Operation
//  Returns:
//  Parameters:
//      handle     An open session handle
//      setMode    Set, reserve, or release ?
//      rowData    Attribute values to set
*F*/

DmiErrorStatus_t  DMI_API
DmiSetMultiple (
    [in]  DmiHandle_t         handle,
    [in]  DmiSetMode_t        setMode,
    [in]  DmiMultiRowData_t*  rowData );


} /* interface dmi_server */
```

INDICATION DELIVERY INTERFACE (CLIENT.IDL)

```
/*M*
//
//  RCS:
//      $Workfile:  client.idl $
//      $Revision:  2.0        $
//      $Modtime:   3/27/96    $
//      $Author:    DMTF       $
//
//  Purpose:
//
//      Describe the DMTF's Management Interface in an IDL that is
//      suitable for building remote management using the DCE-RPC
//      client/server model.  This file, along with client.acf, is
//      compiled with the IDL compiler to produce the following
//      files:
//
//                client.h          C-style interface header file
//                client_c.c        Stub code for the managed system
//                client_s.c        Stub code for the managing application
//
//  Contents:
//
//      The following information is described in version 2.0
//      of the Desktop Management Interface Specification.
//
//  Data Structures:
//
//      DmiNodeAddress              Node address for indication originators
//
//  Indication Delivery:
//
//      DmiDeliverEvent             Deliver event data to an application
//      DmiComponentAdded           A component was added to the database
//      DmiComponentDeleted         A component was deleted from the database
//      DmiLanguageAdded            A component language mapping was added
//      DmiLanguageDeleted          A component language mapping was deleted
//      DmiGroupAdded               A group was added to a component
//      DmiGroupDeleted             A group was deleted from a component
//      DmiSubscriptionNotice       Information about an indication subscription

*M*/

[
    uuid(12f1bec0-5c1c-11cf-9a4b-00aa0034b922),
    version(2.0),
    pointer_default(ptr)
]
    interface dmi_client
{

# include "common.idl"


/*****************************************************************
 * DmiNodeAddress
 *****************************************************************/

/*D*
//  Name:       DmiNodeAddress
//  Purpose:    Addressing information for indication originators
//  Context:    Passed to indication delivery functions
//  Fields:
//      address      Transport-dependent node address
//      rpc          Identifies the RPC (DCE, ONC, etc)
//      transport    Identifies the transport (TPC/IP, SPX, etc)
*D*/
```

```
typedef struct DmiNodeAddress {
    DmiString_t*  address;
    DmiString_t*  rpc;
    DmiString_t*  transport;
} DmiNodeAddress_t;


/*********************************************************************
 * DmiDeliverEvent
 *********************************************************************/

/*F*
//  Name:       DmiDeliverEvent
//  Purpose:    Deliver event data to an application
//  Context:    Indication Delivery
//  Returns:
//  Parameters:
//      handle       An opaque ID returned to the application
//      sender       Address of the node delivering the indication
//      language     Language encoding for the indication data
//      compId       Component reporting the event
//      timestamp    Event generation time
//      rowData      Standard and context-specific indication data
*F*/

DmiErrorStatus_t  DMI_API
DmiDeliverEvent (
    [in]  DmiUnsigned_t       handle,
    [in]  DmiNodeAddress_t*   sender,
    [in]  DmiString_t*        language,
    [in]  DmiId_t             compId,
    [in]  DmiTimestamp_t*     timestamp,
    [in]  DmiMultiRowData_t*  rowData );


/*********************************************************************
 * DmiComponentAdded
 *********************************************************************/

/*F*
//  Name:       DmiComponentAdded
//  Purpose:    A component was added to the database
//  Context:    Indication Delivery
//  Returns:
//  Parameters:
//      handle    An opaque ID returned to the application
//      sender    Address of the node delivering the indication
//      info      Information about the component added
*F*/

DmiErrorStatus_t  DMI_API
DmiComponentAdded (
    [in]  DmiUnsigned_t        handle,
    [in]  DmiNodeAddress_t*    sender,
    [in]  DmiComponentInfo_t*  info );


/*********************************************************************
 * DmiComponentDeleted
 *********************************************************************/

/*F*
//  Name:       DmiComponentDeleted
//  Purpose:    A component was deleted from the database
//  Context:    Indication Delivery
//  Returns:
//  Parameters:
//      handle    An opaque ID returned to the application
//      sender    Address of the node delivering the indication
//      compId    Component deleted from the database
*F*/

DmiErrorStatus_t  DMI_API
DmiComponentDeleted (
```

```
    [in]  DmiUnsigned_t      handle,
    [in]  DmiNodeAddress_t*  sender,
    [in]  DmiId_t            compId );


/*********************************************************************
 * DmiLanguageAdded
 *********************************************************************/

/*F*
//  Name:       DmiLanguageAdded
//  Purpose:    A component language mapping was added
//  Context:    Indication Delivery
//  Returns:
//  Parameters:
//      handle      An opaque ID returned to the application
//      sender      Address of the node delivering the indication
//      compId      Component with new language mapping
//      language    language-code|territory-code|encoding
*F*/

DmiErrorStatus_t  DMI_API
DmiLanguageAdded (
    [in]  DmiUnsigned_t      handle,
    [in]  DmiNodeAddress_t*  sender,
    [in]  DmiId_t            compId,
    [in]  DmiString_t*       language );


/*********************************************************************
 * DmiLanguageDeleted
 *********************************************************************/

/*F*
//  Name:       DmiLanguageDeleted
//  Purpose:    A component language mapping was deleted
//  Context:    Indication Delivery
//  Returns:
//  Parameters:
//      handle      An opaque ID returned to the application
//      sender      Address of the node delivering the indication
//      compId      Component with deleted language mapping
//      language    language-code|territory-code|encoding
*F*/

DmiErrorStatus_t  DMI_API
DmiLanguageDeleted (
    [in]  DmiUnsigned_t      handle,
    [in]  DmiNodeAddress_t*  sender,
    [in]  DmiId_t            compId,
    [in]  DmiString_t*       language );


/*********************************************************************
 * DmiGroupAdded
 *********************************************************************/

/*F*
//  Name:       DmiGroupAdded
//  Purpose:    A group was added to a component
//  Context:    Indication Delivery
//  Returns:
//  Parameters:
//      handle    An opaque ID returned to the application
//      sender    Address of the node delivering the indication
//      compId    Component with new group added
//      info      Information about the group added
*F*/

DmiErrorStatus_t  DMI_API
DmiGroupAdded (
    [in]  DmiUnsigned_t      handle,
    [in]  DmiNodeAddress_t*  sender,
    [in]  DmiId_t            compId,
    [in]  DmiGroupInfo_t*    info );
```

```
/*********************************************************************
 * DmiGroupDeleted
 *********************************************************************/

/*F*
//  Name:       DmiGroupDeleted
//  Purpose:    A group was deleted from a component
//  Context:    Indication Delivery
//  Returns:
//  Parameters:
//      handle     An opaque ID returned to the application
//      sender     Address of the node delivering the indication
//      compId     Component with the group deleted
//      groupId    Group deleted from the component
*F*/

DmiErrorStatus_t  DMI_API
DmiGroupDeleted (
    [in]  DmiUnsigned_t      handle,
    [in]  DmiNodeAddress_t*  sender,
    [in]  DmiId_t            compId,
    [in]  DmiId_t            groupId );


/*********************************************************************
 * DmiSubscriptionNotice
 *********************************************************************/

/*F*
//  Name:       DmiSubscriptionNotice
//  Purpose:    Information about an indication subscription
//  Context:    Indication Delivery
//  Returns:
//  Parameters:
//      handle     An opaque ID returned to the application
//      expired    True=expired; False=expiration pending
//      rowData    Row information to identify the subscription
*F*/

DmiErrorStatus_t  DMI_API
DmiSubscriptionNotice (
    [in]  DmiUnsigned_t      handle,
    [in]  DmiNodeAddress_t*  sender,
    [in]  DmiBoolean_t       expired,
    [in]  DmiRowData_t*      rowData );


} /* interface dmi_client */
```

APPENDIX C - ONC RPCGEN

COMMON DATA STRUCTURES (COMMON.X)

```
/*M*
//
//  RCS:
//      $Workfile:  common.x   $
//      $Revision:  2.0        $
//      $Modtime:   3/27/96    $
//      $Author:    DMTF       $
//
//  Purpose:
//
//      Describe data structures and types for the DMTF's Management
//      Interface in an RPCGEN that is suitable for building remote
//      management using the ONC RPC client/server model.  This
//      file is included in the client.x and server.x files.
//
//  Contents:
//
//      The following information is described in version 2.0
//      of the Desktop Management Interface Specification.
//
//  Enumerated Types:
//
//      DmiSetMode                  Define set operations
//      DmiRequestMode              Define sequential access modes
//      DmiStorageType              Define the storage type for an attribute
//      DmiAccessMode               Define the access modes for an attribute
//      DmiDataType                 Define the data types referenced by DmiDataUnion
//      DmiFileDataInfo             Define the DMI mapping file types
//
//  Data Structures:
//
//      DmiTimestamp                Describes the DMI timestamp structure
//      DmiString                   Describes the DMI string representation
//      DmiOctetString              Describes the DMI octet string representation
//      DmiDataUnion                Discriminated union of DMI data types
//      DmiEnumInfo                 Associates an integer value with descriptive text
//      DmiAttributeInfo            Holds information about an attribute
//      DmiAttributeData            Describes an attribute id, type, and value
//      DmiGroupInfo                Holds information about a group
//      DmiComponentInfo            Holds information about a component
//      DmiFileDataInfo             Holds language file type and mapping data
//      DmiClassNameInfo            Holds a group's id and class string
//      DmiRowRequest               Identifies { component, group, row, ids } to get
//      DmiRowData                  Identifies { component, group, row, values } to set
//
//      DmiAttributeIds             Describes a conformant array of DmiId
//      DmiAttributeValues          Describes a conformant array of DmiAttributeData
//      DmiEnumList                 Describes a conformant array of DmiEnumInfo
//      DmiAttributeList            Describes a conformant array of DmiAttributeInfo
//      DmiGroupList                Describes a conformant array of DmiGroupInfo
//      DmiComponentList            Describes a conformant array of DmiComponentInfo
//      DmiFileDataList             Describes a conformant array of DmiFileDataInfo
//      DmiClassNameList            Describes a conformant array of DmiClassNameInfo
//      DmiStringList               Describes a conformant array of DmiString
//      DmiFileTypeList             Describes a conformant array of DmiFileType
//      DmiMultiRowRequest          Describes a conformant array of DmiRowRequest
//      DmiMultiRowData             Describes a conformant array of DmiRowData
*M*/

# ifndef DMI_API
# define DMI_API
# endif
```

```
/*********************************************************************
 * DmiSetMode
 *********************************************************************/

/*D*
//  Name:       DmiSetMode
//  Purpose:    Define set operations
//  Context:    DmiSetAttributes()
//  Fields:
//      DMI_SET        Set data values
//      DMI_RESERVE    Reserve resources for a set operation
//      DMI_RELEASE    Release previously reserved resources
*D*/

enum DmiSetMode {
    DMI_SET,
    DMI_RESERVE,
    DMI_RELEASE
};
typedef enum DmiSetMode DmiSetMode_t;


/*********************************************************************
 * DmiRequestMode
 *********************************************************************/

/*D*
//  Name:       DmiRequestMode
//  Purpose:    Define sequential access modes
//  Context:    Field in DmiRowRequest,
//  Context:    DmiListComponents(), DmiListComponentsByClass(),
//  Context:    DmiListGroups(), DmiListAttributes(),
//  Fields:
//      DMI_UNIQUE    Access the specified item (or table row)
//      DMI_FIRST     Access the first item
//      DMI_NEXT      Access the next item
*D*/

enum DmiRequestMode {
    DMI_UNIQUE,
    DMI_FIRST,
    DMI_NEXT
};
typedef enum DmiRequestMode DmiRequestMode_t;


/*********************************************************************
 * DmiStorageType
 *********************************************************************/

/*D*
//  Name:       DmiStorageType
//  Purpose:    Define the storage type for an attribute
//  Context:    Field in DmiAttributeInfo
//  Fields:
//      MIF_COMMON      Value is from a small set of possibilities
//      MIF_SPECIFIC    Value is from a large set of possibilities
*D*/

enum DmiStorageType {
    MIF_COMMON,
    MIF_SPECIFIC
};
typedef enum DmiStorageType DmiStorageType_t;
```

```
/*********************************************************************
 * DmiAccessMode
 *********************************************************************/

/*D*
//  Name:       DmiAccessMode
//  Purpose:    Define the access modes for an attribute
//  Context:    Field in DmiAttributeInfo
//  Fields:
//      MIF_UNKNOWN        Unknown access mode
//      MIF_READ_ONLY      Read access only
//      MIF_READ_WRITE     Readable and writable
//      MIF_WRITE_ONLY     Write access only
//      MIF_UNSUPPORTED    Attribute is not supported
*D*/

enum DmiAccessMode {
    MIF_UNKNOWN_ACCESS,
    MIF_READ_ONLY,
    MIF_READ_WRITE,
    MIF_WRITE_ONLY,
    MIF_UNSUPPORTED
};
typedef enum DmiAccessMode DmiAccessMode_t;


/*********************************************************************
 * DmiDataType
 *********************************************************************/

/*D*
//  Name:       DmiDataType
//  Purpose:    Define the data types referenced by DmiDataUnion
//  Context:
//  Fields:
//      MIF_DATATYPE_0       RESERVED
//      MIF_COUNTER          32-bit unsigned integer that never decreases
//      MIF_COUNTER64        64-bit unsigned integer that never decreases
//      MIF_GAUGE            32-bit unsigned integer may increase or decrease
//      MIF_DATATYPE_4       RESERVED
//      MIF_INTEGER          32-bit signed integer; no semantics known
//      MIF_INTEGER64        64-bit signed integer; no semantics known
//      MIF_OCTETSTRING      String of n octets, not necessarily displayable
//      MIF_DISPLAYSTRING    Displayable string of n octets
//      MIF_DATATYPE_9       RESERVED
//      MIF_DATATYPE_10      RESERVED
//      MIF_DATE             28-octet displayable string
(yyyymmddHHMMSS.uuuuuu+ooo)
*D*/

enum DmiDataType {
    MIF_DATATYPE_0,
    MIF_COUNTER,
    MIF_COUNTER64,
    MIF_GAUGE,
    MIF_DATATYPE_4,
    MIF_INTEGER,
    MIF_INTEGER64,
    MIF_OCTETSTRING,
    MIF_DISPLAYSTRING,
    MIF_DATATYPE_9,
    MIF_DATATYPE_10,
    MIF_DATE
};
typedef enum DmiDataType DmiDataType_t;


/*
 * Aliases for the standard data types
 */
# define MIF_INT     MIF_INTEGER
# define MIF_INT64   MIF_INTEGER64
```

```
# define MIF_STRING  MIF_DISPLAYSTRING


/*********************************************************************
 * DmiFileType
 *********************************************************************/

/*D*
//  Name:       DmiFileType
//  Purpose:    Define the DMI mapping file types
//  Context:    Field in DmiFileDataInfo
//  Fields:
//      DMI_FILETYPE_0            RESERVED
//      DMI_FILETYPE_1            RESERVED
//      DMI_MIF_FILE_NAME         File data is DMI MIF file name
//      DMI_MIF_FILE_DATA         File data is DMI MIF data
//      SNMP_MAPPING_FILE_NAME    File data is SNMP MAPPING file name
//      SNMP_MAPPING_FILE_DATA    File data is SNMP MAPPING data
//      DMI_GROUP_FILE_NAME       File data is DMI GROUP MIF file name
//      DMI_GROUP_FILE_DATA       File data is DMI GROUP MIF data
//      MS_FILE_NAME              File data is Microsoft-format file name
//      MS_FILE_DATA              File data is Microsoft-format data
*D*/

enum DmiFileType {
    DMI_FILETYPE_0,
    DMI_FILETYPE_1,
    DMI_MIF_FILE_NAME,
    DMI_MIF_FILE_DATA,
    SNMP_MAPPING_FILE_NAME,
    SNMP_MAPPING_FILE_DATA,
    DMI_GROUP_FILE_NAME,
    DMI_GROUP_FILE_DATA,
    MS_FILE_NAME,
    MS_FILE_DATA
};
typedef enum DmiFileType DmiFileType_t;


/*********************************************************************
 * DMI Data Types
 *********************************************************************/

typedef unsigned long   DmiId_t;
typedef unsigned long   DmiHandle_t;
typedef unsigned long   DmiCounter_t;
typedef unsigned long   DmiErrorStatus_t;
typedef unsigned long   DmiCounter64_t[2];
typedef unsigned long   DmiGauge_t;
typedef unsigned long   DmiUnsigned_t;
typedef long            DmiInteger_t;
typedef unsigned long   DmiInteger64_t[2];
typedef unsigned long   DmiBoolean_t;


/*********************************************************************
 * DmiTimestamp
 *********************************************************************/

/*D*
//  Name:       DmiTimestamp
//  Purpose:    Describes the DMI timestamp structure
//  Context:    Field in DmiDataUnion
//  Fields:
//      year            The year ('1996')
//      month           The month ('1'..'12')
//      yay             The day of the month ('1'..'23')
//      hour            The hour ('0'..'23')
//      minutes         The minutes ('0'..'59')
//      seconds         The seconds ('0'..'60'); includes leap seconds
//      dot             A dot ('.')
//      microSeconds    Microseconds ('0'..'999999')
//      plusODMI Version 2nus     '+' for east, or '-' west of UTC
```

```
//      utcOffset       Minutes ('0'..'720') from UTC
//      padding         Unused padding for 4-byte alignment
*D*/

struct DmiTimestamp {
    char  year          [4];
    char  month         [2];
    char  day           [2];
    char  hour          [2];
    char  minutes       [2];
    char  seconds       [2];
    char  dot;
    char  microSeconds  [6];
    char  plusODMI Version 2nus;
    char  utcOffset     [3];
    char  padding       [3];
};
typedef struct DmiTimestamp DmiTimestamp_t;


/**********************************************************************
 * DmiString
 **********************************************************************/

/*D*
//  Name:       DmiString
//  Purpose:    Describes the DMI string representation
//  Context:    Field in DmiDataUnion
//  Fields:
//      size    Number of octets in the string body
//      body    String contents
//
//  Notes:      For displaystrings, the string is null teDMI Version 2nated,
//              and the null character is included in the size.
*D*/

struct DmiString {
    char  body<>;
};
typedef struct DmiString DmiString_t;
typedef DmiString_t* DmiStringPtr_t;


/**********************************************************************
 * DmiOctetString
 **********************************************************************/

/*D*
//  Name:       DmiOctetString
//  Purpose:    Describes the DMI octet string representation
//  Context:    Field in DmiDataUnion
//  Fields:
//      size    Number of octets in the string body
//      body    String contents
*D*/

struct DmiOctetString {
    char  body<>;
};
typedef struct DmiOctetString DmiOctetString_t;


/**********************************************************************
 * DmiDataUnion
 **********************************************************************/

/*D*
//  Name:       DmiDataUnion
//  Purpose:    Discriminated union of DMI data types
//  Context:    Field in DmiAttributeData
//  Fields:
//      type    Discriminator for the union
//      value   Union of DMI attribute data types
```

```
*D*/

union DmiDataUnion switch (DmiDataType_t type) {
    case MIF_COUNTER:       DmiCounter_t       counter;
    case MIF_COUNTER64:     DmiCounter64_t     counter64;
    case MIF_GAUGE:         DmiGauge_t         gauge;
    case MIF_INTEGER:       DmiInteger_t       integer;
    case MIF_INTEGER64:     DmiInteger64_t     integer64;
    case MIF_OCTETSTRING:   DmiOctetString_t*  octetstring;
    case MIF_DISPLAYSTRING: DmiString_t*       str;
    case MIF_DATE:          DmiTimestamp_t*    date;
};
typedef union DmiDataUnion DmiDataUnion_t;


/**********************************************************************
 * DmiEnumInfo
 **********************************************************************/

/*D*
//  Name:       DmiEnumInfo
//  Purpose:    Associates an integer value with descriptive text
//  Context:    Element in DmiEnumList
//  Fields:
//      name    Enumeration name
//      value   Enumeration value
*D*/

struct DmiEnumInfo {
    DmiString_t*    name;
    DmiInteger_t    value;
};
typedef struct DmiEnumInfo DmiEnumInfo_t;


/**********************************************************************
 * DmiAttributeInfo
 **********************************************************************/

/*D*
//  Name:       DmiAttributeInfo
//  Purpose:    Holds information about an attribute
//  Context:    Element in DmiAttributeList
//  Fields:
//      id              Attribute ID
//      name            Attribute name string
//      pragma          Attribute pragma string           [optional]
//      description     Attribute description string      [optional]
//      storage         Common or specific storage
//      access          Readonly, read-write, etc
//      type            Counter, integer, etc
//      maxSize         Maximum length of the attribute
//      enumList        EnumList for enumerated types     [optional]
*D*/

struct DmiAttributeInfo {
    DmiId_t                 id;
    DmiString_t*            name;
    DmiString_t*            pragma;
    DmiString_t*            description;
    DmiStorageType_t        storage;
    DmiAccessMode_t         access;
    DmiDataType_t           type;
    DmiUnsigned_t           maxSize;
    struct DmiEnumList*     enumList;
};
typedef struct DmiAttributeInfo DmiAttributeInfo_t;
```

```
/*********************************************************************
 * DmiAttributeData
 *********************************************************************/

/*D*
//  Name:       DmiAttributeData
//  Purpose:    Describes an attribute id, type, and value
//  Context:    Element in DmiAttributeValues
//  Fields:
//      id      Attribute ID
//      data    Attribute type and value
*D*/

struct DmiAttributeData {
    DmiId_t           id;
    DmiDataUnion_t    data;
};
typedef struct DmiAttributeData DmiAttributeData_t;


/*********************************************************************
 * DmiGroupInfo
 *********************************************************************/

/*D*
//  Name:       DmiGroupInfo
//  Purpose:    Holds information about a group
//  Context:    Element in DmiGroupList
//  Fields:
//      id              Group ID
//      name            Group name string
//      pragma          Group pragma string                 [optional]
//      className       Group class name string
//      description     Group description string            [optional]
//      keyList         Attribute IDs for table row keys    [optional]
*D*/

struct DmiGroupInfo {
    DmiId_t                   id;
    DmiString_t*              name;
    DmiString_t*              pragma;
    DmiString_t*              className;
    DmiString_t*              description;
    struct DmiAttributeIds*   keyList;
};
typedef struct DmiGroupInfo DmiGroupInfo_t;


/*********************************************************************
 * DmiComponentInfo
 *********************************************************************/

/*D*
//  Name:       DmiComponentInfo
//  Purpose:    Holds information about a component
//  Context:    Element in DmiComponentList
//  Fields:
//      id              Component ID
//      name            Component name string
//      pragma          Component pragma string         [optional]
//      description     Component description string    [optional]
//      exactMatch
//         idl_true  = Exact match
//         idl_false = Possible match
*D*/

struct DmiComponentInfo {
    DmiId_t         id;
    DmiString_t*    name;
    DmiString_t*    pragma;
    DmiString_t*    description;
    DmiBoolean_t    exactMatch;
```

```
};
typedef struct DmiComponentInfo DmiComponentInfo_t;


/**********************************************************************
 * DmiFileDataInfo
 **********************************************************************/

/*D*
//  Name:       DmiFileDataInfo
//  Purpose:    Holds language file type and mapping data
//  Context:    Element in DmiFileDataList
//  Fields:
//      fileType    MIF file, SNMP mapping file, etc
//      fileData    The file info (name -or- contents)
*D*/

struct DmiFileDataInfo {
    DmiFileType_t        fileType;
    DmiOctetString_t*    fileData;
};
typedef struct DmiFileDataInfo DmiFileDataInfo_t;


/**********************************************************************
 * DmiClassNameInfo
 **********************************************************************/

/*D*
//  Name:       DmiClassNameInfo
//  Purpose:    Holds a group's id and class string
//  Context:    Element in DmiClassNameList
//  Fields:
//      id           Group ID
//      className    Group class name string
*D*/

     struct DmiClassNameInfo {
    DmiId_t       id;
    DmiString_t*  className;
};
typedef struct DmiClassNameInfo DmiClassNameInfo_t;


/**********************************************************************
 * DmiRowRequest
 **********************************************************************/

/*D*
//  Name:       DmiRowRequest
//  Purpose:    Identifies { component, group, row, ids } to get
//  Context:    Element in DmiMultiRowRequest
//  Fields:
//      compId         Component ID
//      groupId        Group ID
//      requestMode    Get from specified row, first row, or next row
//      keyList        Array of values for key attributes
//      ids            Array of IDs for data attributes
*D*/

struct DmiRowRequest {
    DmiId_t                       compId;
    DmiId_t                       groupId;
    DmiRequestMode_t              requestMode;
    struct DmiAttributeValues*    keyList;
    struct DmiAttributeIds*       ids;
};
typedef struct DmiRowRequest DmiRowRequest_t;
```

```
/*********************************************************************
 * DmiRowData
 *********************************************************************/

/*D*
//  Name:       DmiRowData
//  Purpose:    Identifies { component, group, row, values } to set
//  Context:    Element in DmiMultiRowData
//  Fields:
//      compId       Component ID
//      groupId      Group ID
//      className    Group class name for events, or 0    [optional]
//      keyList      Array of values for key attributes
//      values       Array of values for data attributes
//
//  Notes:      This structure is used for setting attributes, getting
//              attributes, and for providing indication data.  The
//              className string is only required when returning
//              indication data.  For other uses, the field can be 0.
*D*/

struct DmiRowData {
    DmiId_t                      compId;
    DmiId_t                      groupId;
    DmiString_t*                 className;
    struct DmiAttributeValues*   keyList;
    struct DmiAttributeValues*   values;
};
typedef struct DmiRowData DmiRowData_t;


/*********************************************************************
 * DmiAttributeIds
 *********************************************************************/

/*D*
//  Name:       DmiAttributeIds
//  Purpose:    Describes a conformant array of DmiId
//  Context:    Field in DmiRowRequest
//  Fields:
//      size    Array elements
//      list    Array data
*D*/

struct DmiAttributeIds {
    DmiId_t  list<>;
};
typedef struct DmiAttributeIds DmiAttributeIds_t;


/*********************************************************************
 * DmiAttributeValues
 *********************************************************************/

/*D*
//  Name:       DmiAttributeValues
//  Purpose:    Describes a conformant array of DmiAttributeData
//  Context:    Field in DmiRowRequest, DmiRowData
//  Fields:
//      size    Array elements
//      list    Array data
*D*/

struct DmiAttributeValues {
    DmiAttributeData_t  list<>;
};
typedef struct DmiAttributeValues DmiAttributeValues_t;
```

```
/*****************************************************************
 * DmiEnumList
 *****************************************************************/

/*D*
//  Name:       DmiEnumList
//  Purpose:    Describes a conformant array of DmiEnumInfo
//  Context:    DmiEnumAttributes()
//  Fields:
//      size    Array elements
//      list    Array data
*D*/

struct DmiEnumList {
    DmiEnumInfo_t  list<>;
};
typedef struct DmiEnumList DmiEnumList_t;


/*****************************************************************
 * DmiAttributeList
 *****************************************************************/

/*D*
//  Name:       DmiAttributeList
//  Purpose:    Describes a conformant array of DmiAttributeInfo
//  Context:    DmiListAttributes()
//  Fields:
//      size    Array elements
//      list    Array data
*D*/

struct DmiAttributeList {

    DmiAttributeInfo_t  list<>;
};
typedef struct DmiAttributeList DmiAttributeList_t;


/*****************************************************************
 * DmiGroupList
 *****************************************************************/

/*D*
//  Name:       DmiGroupList
//  Purpose:    Describes a conformant array of DmiGroupInfo
//  Context:    DmiListGroups()
//  Fields:
//      size    Array elements
//      list    Array data
*D*/

struct DmiGroupList {
    DmiGroupInfo_t  list<>;
};
typedef struct DmiGroupList DmiGroupList_t;


/*****************************************************************
 * DmiComponent
 *****************************************************************/

/*D*
//  Name:       DmiComponentList
//  Purpose:    Describes a conformant array of DmiComponentInfo
//  Context:    DmiListComponents(), DmiListComponentsByClass()
//  Fields:
//      size    Array elements
//      list    Array data
*D*/
```

```
struct DmiComponentList {
    DmiComponentInfo_t  list<>;
};
typedef struct DmiComponentList DmiComponentList_t;


/*********************************************************************
 * DmiFileDataList
 *********************************************************************/

/*D*
//  Name:       DmiFileDataList
//  Purpose:    Describes a conformant array of DmiFileDataInfo
//  Context:    DmiAddComponent(), DmiAddLanguage(), DmiAddGroup()
//  Fields:
//      size    Array elements
//      list    Array data
*D*/

struct DmiFileDataList {
    DmiFileDataInfo_t  list<>;
};
typedef struct DmiFileDataList DmiFileDataList_t;


/*********************************************************************
 * DmiClassNameList
 *********************************************************************/

/*D*
//  Name:       DmiClassNameList
//  Purpose:    Describes a conformant array of DmiClassNameInfo
//  Context:    DmiListClassNames()
//  Fields:
//      size    Array elements
//      list    Array data
*D*/

struct DmiClassNameList {
    DmiClassNameInfo_t  list<>;
};
typedef struct DmiClassNameList DmiClassNameList_t;


/*********************************************************************
 * DmiStringList
 *********************************************************************/

/*D*
//  Name:       DmiStringList
//  Purpose:    Describes a conformant array of DmiStrings
//  Context:    DmiListLanguages()
//  Fields:
//      size    Array elements
//      list    Array data
*D*/

struct DmiStringList {
    DmiStringPtr_t  list<>;
};
typedef struct DmiStringList DmiStringList_t;


/*********************************************************************
 * DmiFileTypeList
 *********************************************************************/

/*D*
//  Name:       DmiFileTypeList
//  Purpose:    Describes a conformant array of DmiFileType entries
//  Context:    DmiGetVersion()
//  Fields:
```

```
//      size    Array elements
//      list    Array data
*D*/

struct DmiFileTypeList {
    DmiFileType_t  list<>;
};
typedef struct DmiFileTypeList DmiFileTypeList_t;


/*********************************************************************
 * DmiMultiRowRequest
 *********************************************************************/

/*D*
//  Name:       DmiMultiRowRequest
//  Purpose:    Describes a conformant array of DmiRowRequest
//  Context:    DmiGetAttributes()
//  Fields:
//      size    Array elements
//      list    Array data
*D*/

struct DmiMultiRowRequest {
    DmiRowRequest_t  list<>;
};
typedef struct DmiMultiRowRequest DmiMultiRowRequest_t;


/*********************************************************************
 * DmiMultiRowData
 *********************************************************************/

/*D*
//  Name:       DmiMultiRowData
//  Purpose:    Describes a conformant array of DmiRowData
//  Context:    DmiGetAttributes(), DmiSetAttributes()
//  Fields:
//      size    Array elements
//      list    Array data
*D*/

struct DmiMultiRowData {
    DmiRowData_t  list<>;
};
typedef struct DmiMultiRowData DmiMultiRowData_t;
```

MANAGEMENT INTERFACE (SERVER.X)

```
/*M*
//
//  RCS:
//      $Workfile:  server.x   $
//      $Revision:  2.0        $
//      $Modtime:   3/27/96    $
//      $Author:    DMTF       $
//
//  Purpose:
//
//      Describe the DMTF's Management Interface in an RPCGEN that is
//      suitable for building remote management using the ONC RPC
//      client/server model.  This file is compiled with the RPCGEN
//      compiler to produce the following files:
//
//                server.h          C-style interface header file
//                server_c.c        Stub code for the rmi client
//                server_s.c        Stub code for the rmi server
//
//  Contents:
//
//      The following information is described in version 2.0
//      of the Desktop Management Interface Specification.
//
//  Initialization:
//
//      DmiRegister                 Register a session with a remote system
//      DmiUnregister               Unregister a previously registered session
//      DmiGetVersion               Get DMI Service Provider version information
//      DmiGetConfig                Get session configuration parameters
//      DmiSetConfig                Set session configuration parameters
//
//  Discovery:
//
//      DmiListComponents           List component properties
//      DmiListComponentsByClass    List components matching certain criteria
//      DmiListLanguages            List a component's language strings
//      DmiListClassNames           List a component's class names and group ids
//      DmiListGroups               List group properties
//      DmiListAttributes           List attribute properties
//
//  Operation:
//
//      DmiAddRow                   Add a new row to a table
//      DmiDeleteRow                Delete a row from a table
//      DmiGetMultiple            Get a collection of attribute values
//      DmiSetMultiple            Set a collection of attribute values
//
//  Database Administration [optional]:
//
//      DmiAddComponent             Add a new component to the DMI database
//      DmiAddLanguage              Add a new language mapping for a component
//      DmiAddGroup                 Add a new group to a component
//      DmiDeleteComponent          Delete a component from the DMI database
//      DmiDeleteLanguage           Delete a language mapping for a component
//      DmiDeleteGroup              Delete a group from a component
*M*/

# include "common.x"
```

```
/*****************************************************************
 * DmiRegister
 *****************************************************************/

/*F*
//  Name:       DmiRegister
//  Purpose:    Register a session with a remote system
//  Context:    Initialization
//  Returns:
//  Parameters:
//      handle    On completion, an open session handle
//
//  Notes:      The client provides the address of the handle
//              parameter and the server fills it in.  All commands
//              except DmiRegister() require a valid handle, so
//              this must be the first command sent to the DMI server.
*F*/

struct DmiRegisterIN {
    DmiHandle_t   handle;
};

struct DmiRegisterOUT {
    DmiErrorStatus_t error_status;
    DmiHandle_t*  handle;
};


/*****************************************************************
 * DmiUnregister
 *****************************************************************/

/*F*
//  Name:       DmiUnregister
//  Purpose:    Unregister a previously registered session
//  Context:    Initialization
//  Returns:
//  Parameters:
//      handle    An open session handle to be closed
*F*/

struct DmiUnregisterOUT {
    DmiErrorStatus_t error_status;
};

struct DmiUnregisterIN {
    DmiHandle_t  handle;
};


/*****************************************************************
 * DmiGetVersion
 *****************************************************************/

/*F*
//  Name:       DmiGetVersion
//  Purpose:    Get DMI Service Provider version information
//  Context:    Initialization
//  Returns:
//  Parameters:
//      handle           An open session handle
//      dmiSpecLevel     The DMI Specification version
//      description      The OS-specific Service Provider version
//      fileTypes        The file types supported for MIF installation
//
//  Notes:      1. The client must free the dmiSpecLevel string
//              2. The client must free the description string
*F*/

struct DmiGetVersionOUT {
    DmiErrorStatus_t error_status;
    DmiString_t*        dmiSpecLevel;
```

```
    DmiString_t*        description;
    DmiFileTypeList_t*  fileTypes;
};

struct DmiGetVersionIN {
    DmiHandle_t          handle;
};


/*************************************************************************
 * DmiGetConfig
 *************************************************************************/

/*F*
//  Name:       DmiGetConfig
//  Purpose:    Get session configuration parameters
//  Context:    Initialization
//  Returns:
//  Parameters:
//      handle      An open session handle
//      language    language-code|territory-code|encoding
//
//  Notes:      The client must free the language string
*F*/

struct DmiGetConfigOUT {
     DmiErrorStatus_t  error_status;
    DmiString_t*  language;
};

struct DmiGetConfigIN {
    DmiHandle_t    handle;
};


/*************************************************************************
 * DmiSetConfig
 *************************************************************************/

/*F*
//  Name:       DmiSetConfig
//  Purpose:    Set session configuration parameters
//  Context:    Initialization
//  Returns:
//  Parameters:
//      handle      An open session handle
//      language    language-code|territory-code|encoding
*F*/

struct DmiSetConfigOUT {
     DmiErrorStatus_t  error_status;
};

struct DmiSetConfigIN {
    DmiHandle_t   handle;
    DmiString_t*  language;
};


/*************************************************************************
 * DmiListComponents
 *************************************************************************/

/*F*
//  Name:       DmiListComponents
//  Purpose:    List component properties
//  Context:    Discovery
//  Returns:
//  Parameters:
//      handle            An open session handle
//      requestMode       Unique, first, or next component ?
//      maxCount          Maximum number to return, or 0 for all
//      getPragma         Get optional pragma string ?
```

```
//      getDescription    Get optional component description ?
//      compId            Component to start with (see requestMode)
//      reply             List of components
//
//  Notes:      The client must free the reply structure
*F*/

struct DmiListComponentsOUT {
     DmiErrorStatus_t  error_status;
    DmiComponentList_t*  reply;
};

struct DmiListComponentsIN {
    DmiHandle_t           handle;
    DmiRequestMode_t      requestMode;
    DmiUnsigned_t         maxCount;
    DmiBoolean_t          getPragma ;
    DmiBoolean_t          getDescription;
    DmiId_t               compId;
};


/*********************************************************************
 * DmiListComponentsByClass
 *********************************************************************/

/*F*
//  Name:       DmiListComponentsByClass
//  Purpose:    List components matching certain criteria
//  Context:    Discovery
//  Returns:
//  Parameters:
//      handle            An open session handle
//      requestMode       Unique, first, or next component ?
//      maxCount          Maximum number to return, or 0 for all
//      getPragma         Get optional pragma string ?
//      getDescription    Get optional component description ?
//      compId            Component to start with (see requestMode)
//      className         Group class name string to match
//      keyList           Group row keys to match, or null
//      reply             List of components
//
//  Notes:      The client must free the reply structure
*F*/

struct DmiListComponentsByClassOUT {
     DmiErrorStatus_t  error_status;
    DmiComponentList_t*   reply;
};

struct DmiListComponentsByClassIN {
    DmiHandle_t            handle;
    DmiRequestMode_t       requestMode;
    DmiUnsigned_t          maxCount;
    DmiBoolean_t           getPragma;
    DmiBoolean_t           getDescription;
    DmiId_t                compId;
    DmiString_t*           className;
    DmiAttributeValues_t*  keyList;
};


/*********************************************************************
 * DmiListLanguages
 *********************************************************************/

/*F*
//  Name:       DmiListLanguages
//  Purpose:    List a component's language strings
//  Context:    Discovery
//  Returns:
//  Parameters:
//      handle         An open session handle
```

```
//      maxCount       Maximum number to return, or 0 for all
//      compId         Component to access
//      reply          List of language strings
//
//  Notes:      The client must free the reply structure
*F*/

struct DmiListLanguagesOUT {
     DmiErrorStatus_t  error_status;
    DmiStringList_t*  reply;
};

struct DmiListLanguagesIN {
    DmiHandle_t        handle;
    DmiUnsigned_t      maxCount;
    DmiId_t            compId;
};


/*********************************************************************
 * DmiListClassNames
 *********************************************************************/

/*F*
//  Name:       DmiListClassNames
//  Purpose:    List a component's class names and group ids
//  Context:    Discovery
//  Returns:
//  Parameters:
//      handle         An open session handle
//      maxCount       Maximum number to return, or 0 for all
//      compId         Component to access
//      reply          List of class names and group ids
//
//  Notes:      The client must free the reply structure
*F*/

struct DmiListClassNamesOUT {
     DmiErrorStatus_t  error_status;
    DmiClassNameList_t*  reply;
};

struct DmiListClassNamesIN {
    DmiHandle_t           handle;
    DmiUnsigned_t         maxCount;
    DmiId_t               compId;
};


/*********************************************************************
 * DmiListGroups
 *********************************************************************/

/*F*
//  Name:       DmiListGroups
//  Purpose:    List group properties
//  Context:    Discovery
//  Returns:
//  Parameters:
//      handle            An open session handle
//      requestMode       Unique, first, or next group ?
//      maxCount          Maximum number to return, or 0 for all
//      getPragma         Get optional pragma string ?
//      getDescription    Get optional group description ?
//      compId            Component to access
//      groupId           Group to start with (see requestMode)
//      reply             List of groups
//
//  Notes:      The client must free the reply structure
*F*/

struct DmiListGroupsOUT {
     DmiErrorStatus_t  error_status;
```

```
    DmiGroupList_t*   reply;
};

struct DmiListGroupsIN {
    DmiHandle_t        handle;
    DmiRequestMode_t   requestMode;
    DmiUnsigned_t      maxCount;
    DmiBoolean_t       getPragma;
    DmiBoolean_t       getDescription;
    DmiId_t            compId;
    DmiId_t            groupId;
};


/*********************************************************************
 * DmiListAttributes
 *********************************************************************/

/*F*
//  Name:       DmiListAttributes
//  Purpose:    List attribute properties
//  Context:    Discovery
//  Returns:
//  Parameters:
//      handle            An open session handle
//      requestMode       Unique, first, or next attribute ?
//      maxCount          Maximum number to return, or 0 for all
//      getPragma         Get optional pragma string ?
//      getDescription    Get optional attribute description ?
//      compId            Component to access
//      groupId           Group to access
//      attribId          Attribute to start with (see requestMode)
//      reply             List of attributes
//
//  Notes:      The client must free the reply structure
*F*/

struct DmiListAttributesOUT {
     DmiErrorStatus_t  error_status;
    DmiAttributeList_t*  reply;
};

struct DmiListAttributesIN {
    DmiHandle_t           handle;
    DmiRequestMode_t      requestMode;
    DmiUnsigned_t         maxCount;
    DmiBoolean_t          getPragma;
    DmiBoolean_t          getDescription;
    DmiId_t               compId;
    DmiId_t               groupId;
    DmiId_t               attribId;
};


/*********************************************************************
 * DmiAddComponent
 *********************************************************************/

/*F*
//  Name:       DmiAddComponent
//  Purpose:    Add a new component to the DMI database
//  Context:    Database Administration
//  Returns:
//  Parameters:
//      handle      An open session handle
//      fileData    MIF file data for the component
//      compId      On completion, the SP-allocated component I
//      errors      Installation error messages
*F*/

struct DmiAddComponentOUT {
     DmiErrorStatus_t  error_status;
    DmiId_t            compId;
```

```
    DmiStringList_t*   errors;
};

struct DmiAddComponentIN {
    DmiHandle_t         handle;
    DmiFileDataList_t*  fileData;
};


/*************************************************************************
 * DmiAddLanguage
 ************************************************************************/

/*F*
//  Name:       DmiAddLanguage
//  Purpose:    Add a new language mapping for a component
//  Context:    Database Administration
//  Returns:
//  Parameters:
//      handle      An open session handle
//      fileData    Language mapping file for the component
//      compId      Component to access
//      errors      Installation error messages
*F*/

struct DmiAddLanguageOUT {
     DmiErrorStatus_t  error_status;
    DmiStringList_t*   errors;
};

struct DmiAddLanguageIN {
    DmiHandle_t         handle;
    DmiFileDataList_t*  fileData;
    DmiId_t             compId;
};


/*************************************************************************
 * DmiAddGroup
 ************************************************************************/

/*F*
//  Name:       DmiAddGroup
//  Purpose:    Add a new group to a component
//  Context:    Database Administration
//  Returns:
//  Parameters:
//      handle      An open session handle
//      fileData    MIF file data for the group definition
//      compId      Component to access
//      groupId     On completion, the SP-allocated group ID
//      errors      Installation error messages
*F*/

struct DmiAddGroupOUT {
     DmiErrorStatus_t   error_status;
    DmiId_t             groupId;
    DmiStringList_t*   errors;
};

struct DmiAddGroupIN {
    DmiHandle_t         handle;
    DmiFileDataList_t*  fileData;
    DmiId_t             compId;
};


/*************************************************************************
 * DmiDeleteComponent
 ************************************************************************/

/*F*
//  Name:       DmiDeleteComponent
```

```
//  Purpose:    Delete a component from the DMI database
//  Context:    Database Administration
//  Returns:
//  Parameters:
//      handle    An open session handle
//      compId    Component to delete
*F*/

struct DmiDeleteComponentOUT {
    DmiErrorStatus_t   error_status;
};

struct DmiDeleteComponentIN {
    DmiHandle_t  handle;
    DmiId_t      compId;
};


/*********************************************************************
 * DmiDeleteLanguage
 *********************************************************************/

/*F*
//  Name:       DmiDeleteLanguage
//  Purpose:    Delete a language mapping for a component
//  Context:    Database Administration
//  Returns:
//  Parameters:
//      handle      An open session handle
//      language    language-code|territory-code|encoding
//      compId      Component to access
*F*/

struct DmiDeleteLanguageOUT {
    DmiErrorStatus_t  error_status;
};

struct DmiDeleteLanguageIN {
    DmiHandle_t   handle;
    DmiString_t*  language;
    DmiId_t       compId;
};


/*********************************************************************
 * DmiDeleteGroup
 *********************************************************************/

/*F*
//  Name:       DmiDeleteGroup
//  Purpose:    Delete a group from a component
//  Context:    Database Administration
//  Returns:
//  Parameters:
//      handle     An open session handle
//      compId     Component containing group
//      groupId    Group to delete
*F*/

struct DmiDeleteGroupOUT {
    DmiErrorStatus_t   error_status;
};

struct DmiDeleteGroupIN {
    DmiHandle_t  handle;
    DmiId_t      compId;
    DmiId_t      groupId;
};
```

```
/*********************************************************************
 * DmiAddRow
 *********************************************************************/

/*F*
//  Name:       DmiAddRow
//  Purpose:    Add a new row to a table
//  Context:    Operation
//  Returns:
//  Parameters:
//      handle    An open session handle
//      rowData   Attribute values to set
*F*/

struct DmiAddRowOUT {
     DmiErrorStatus_t  error_status;
};

struct DmiAddRowIN {
    DmiHandle_t    handle;
    DmiRowData_t*  rowData;
};


/*********************************************************************
 * DmiDeleteRow
 *********************************************************************/

/*F*
//  Name:       DmiDeleteRow
//  Purpose:    Delete a row from a table
//  Context:    Operation
//  Returns:
//  Parameters:
//      handle     An open session handle
//      rowData    Row { component, group, key } to delete
*F*/

struct DmiDeleteRowOUT {
     DmiErrorStatus_t error_status;
};

struct DmiDeleteRowIN {
    DmiHandle_t    handle;
    DmiRowData_t*  rowData;
};


/*********************************************************************
 * DmiGetMultiple
 *********************************************************************/

/*F*
//  Name:       DmiGetMultiple
//  Purpose:    Get a collection of attribute values
//  Context:    Operation
//  Returns:
//  Parameters:
//      handle     An open session handle
//      request    Attributes to get
//      rowData    Requested attribute values
//
//  Notes:      1. The request may be for a SINGLE row (size = 1)
//              2. An empty id list for a row means "get all attributes"
//              3. The client must free the rowData structure
*F*/
```

```
struct DmiGetMultipleOUT {
     DmiErrorStatus_t  error_status;
    DmiMultiRowData_t*    rowData;
};

struct DmiGetMultipleIN {
    DmiHandle_t            handle;
    DmiMultiRowRequest_t*  request;
};


/*********************************************************************
 * DmiSetMultiple
 *********************************************************************/

/*F*
//  Name:       DmiSetMultiple
//  Purpose:    Set a collection of attributes
//  Context:    Operation
//  Returns:
//  Parameters:
//      handle     An open session handle
//      setMode    Set, reserve, or release ?
//      rowData    Attribute values to set
*F*/

struct DmiSetMultipleOUT {
     DmiErrorStatus_t  error_status;
};

struct DmiSetMultipleIN {
    DmiHandle_t         handle;
    DmiSetMode_t        setMode;
    DmiMultiRowData_t*  rowData;
};


/*********************************************************************
 * DmiGetAttribute
*********************************************************************/

/*F*
//  Name:       DmiGetAttribute
//  Purpose:    Get a single attribute value
//  Context:    Operation
//  Returns:
//  Parameters:
//      handle      An open session handle
//      compId      Component to access
//      groupId     Group within component
//      attribId    Attribute within group
//      keyList     Keylist to specify a table row    [optional]
//      value       Attribute value returned
*F*/

struct DmiGetAttributeOUT {
     DmiErrorStatus_t   error_status;
    DmiDataUnion_t*    value;
};

struct DmiGetAttributeIN {
    DmiHandle_t            handle;
    DmiId_t                compId;
     DmiId_t                groupId;
     DmiId_t                attribId;
     DmiAttributeValues_t*  keyList;
};
```

```
/*********************************************************************
 * DmiSetAttribute
 *********************************************************************/

/*F*
//  Name:       DmiSetAttribute
//  Purpose:    Set a single attribute value
//  Context:    Operation
//  Returns:
//  Parameters:
//      handle      An open session handle
//      compId      Component to access
//      groupId     Group within component
//      attribId    Attribute within group
//      keyList     Keylist to specify a table row    [optional]
//      setMode     Set, reserve, or release ?
//      value       Attribute value to set
*F*/
struct DmiSetAttributeOUT {
    DmiErrorStatus_t   error_status;
};

struct DmiSetAttributeIN {
    DmiHandle_t            handle;
    DmiId_t                compId;
    DmiId_t                groupId;
    DmiId_t                attribId;
    DmiAttributeValues_t*  keyList;
    DmiSetMode_t           setMode;
    DmiDataUnion_t*        value;
};


program DMI2_SERVER {
    version DMI2_SERVER_VERSION {
        DmiRegisterOUT _DmiRegister ( DmiRegisterIN ) = 0x200;
        DmiUnregisterOUT _DmiUnregister ( DmiUnregisterIN ) = 0x201;
        DmiGetVersionOUT _DmiGetVersion ( DmiGetVersionIN ) = 0x202;
        DmiGetConfigOUT _DmiGetConfig ( DmiGetConfigIN ) = 0x203;
        DmiSetConfigOUT _DmiSetConfig ( DmiSetConfigIN ) = 0x204;
        DmiListComponentsOUT _DmiListComponents ( DmiListComponentsIN ) = 0x205;
        DmiListComponentsByClassOUT _DmiListComponentsByClass (
        DmiListComponentsByClassIN ) = 0x206;
        DmiListLanguagesOUT _DmiListLanguages ( DmiListLanguagesIN ) = 0x207;
        DmiListClassNamesOUT _DmiListClassNames ( DmiListClassNamesIN ) = 0x208;
        DmiListGroupsOUT _DmiListGroups ( DmiListGroupsIN ) = 0x209;
        DmiListAttributesOUT _DmiListAttributes ( DmiListAttributesIN ) = 0x20a;
        DmiAddRowOUT _DmiAddRow ( DmiAddRowIN ) = 0x20b;
        DmiDeleteRowOUT _DmiDeleteRow ( DmiDeleteRowIN ) = 0x20c;
        DmiGetMultipleOUT _DmiGetMultiple ( DmiGetMultipleIN ) = 0x20d;
        DmiSetMultipleOUT _DmiSetMultiple ( DmiSetMultipleIN ) = 0x20e;
        DmiAddComponentOUT _DmiAddComponent ( DmiAddComponentIN ) = 0x20f;
        DmiAddLanguageOUT _DmiAddLanguage ( DmiAddLanguageIN ) = 0x210;
        DmiAddGroupOUT _DmiAddGroup ( DmiAddGroupIN ) = 0x211;
        DmiDeleteComponentOUT _DmiDeleteComponent ( DmiDeleteComponentIN ) =
        0x212;
        DmiDeleteLanguageOUT _DmiDeleteLanguage ( DmiDeleteLanguageIN ) = 0x213;
        DmiDeleteGroupOUT _DmiDeleteGroup ( DmiDeleteGroupIN ) = 0x214;
        DmiGetAttributeOUT _DmiGetAttribute ( DmiGetAttributeIN ) = 0x215;
        DmiSetAttributeOUT _DmiSetAttribute ( DmiSetAttributeIN ) = 0x216;
    } = 1;
} = 300598;
```

INDICATION DELIVERY INTERFACE (CLIENT.X)

```
/*M*
//
//  RCS:
//      $Workfile:  client.  x $
//      $Revision:  2.0        $
//      $Modtime:   3/27/96    $
//      $Author:    DMTF       $
//
//  Purpose:
//
//      Describe the DMTF's Management Interface in an RPCGEN that is
//      suitable for building remote management using the ONC RPC
//      client/server model.  This file is compiled with the RPCGEN
//      compiler to produce the following files:
//
//                client.h          C-style interface header file
//                client_c.c        Stub code for the managed system
//                client_s.c        Stub code for the managing application
//
//  Contents:
//
//      The following information is described in version 2.0
//      of the Desktop Management Interface Specification.
//
//  Data Structures:
//
//      DmiNodeAddress              Node address for indication originators
//
//  Indication Delivery:
//
//      DmiDeliverEvent             Deliver event data to an application
//      DmiComponentAdded           A component was added to the database
//      DmiComponentDeleted         A component was deleted from the database
//      DmiLanguageAdded            A component language mapping was added
//      DmiLanguageDeleted          A component language mapping was deleted
//      DmiGroupAdded               A group was added to a component
//      DmiGroupDeleted             A group was deleted from a component
//      DmiSubscriptionNotice       Information about an indication subscription

*M*/

# include "common.x"


/*****************************************************************
 * DmiNodeAddress
 *****************************************************************/

/*D*
//  Name:       DmiNodeAddress
//  Purpose:    Addressing information for indication originators
//  Context:    Passed to indication delivery functions
//  Fields:
//      address      Transport-dependent node address
//      rpc          Identifies the RPC (DCE, ONC, etc)
//      transport    Identifies the transport (TPC/IP, SPX, etc)
*D*/

struct DmiNodeAddress {
    DmiString_t*  address;
    DmiString_t*  rpc;
    DmiString_t*  transport;
};
typedef struct DmiNodeAddress DmiNodeAddress_t;
```

```
/*********************************************************************
 * DmiDeliverEvent
 *********************************************************************/

/*F*
//  Name:       DmiDeliverEvent
//  Purpose:    Deliver event data to an application
//  Context:    Indication Delivery
//  Returns:
//  Parameters:
//      handle       An opaque ID returned to the application
//      sender       Address of the node delivering the indication
//      language     Language encoding for the indication data
//      compId       Component reporting the event
//      timestamp    Event generation time
//      rowData      Standard and context-specific indication data
*F*/

struct DmiDeliverEventIN {
    DmiUnsigned_t       handle;
    DmiNodeAddress_t*   sender;
    DmiString_t*        language;
    DmiId_t             compId;
    DmiTimestamp_t*     timestamp;
    DmiMultiRowData_t*  rowData;
};


/*********************************************************************
 * DmiComponentAdded
 *********************************************************************/

/*F*
//  Name:       DmiComponentAdded
//  Purpose:    A component was added to the database
//  Context:    Indication Delivery
//  Returns:
//  Parameters:
//      handle    An opaque ID returned to the application
//      sender    Address of the node delivering the indication
//      info      Information about the component added
*F*/

struct DmiComponentAddedIN {
    DmiUnsigned_t        handle;
    DmiNodeAddress_t*    sender;
    DmiComponentInfo_t*  info;
};


/*********************************************************************
 * DmiComponentDeleted
 *********************************************************************/

/*F*
//  Name:       DmiComponentDeleted
//  Purpose:    A component was deleted from the database
//  Context:    Indication Delivery
//  Returns:
//  Parameters:
//      handle    An opaque ID returned to the application
//      sender    Address of the node delivering the indication
//      compId    Component deleted from the database
*F*/

struct DmiComponentDeletedIN {
    DmiUnsigned_t      handle;
    DmiNodeAddress_t*  sender;
    DmiId_t            compId;
};
```

```
/*****************************************************************
 * DmiLanguageAdded
 *****************************************************************/

/*F*
//  Name:       DmiLanguageAdded
//  Purpose:    A component language mapping was added
//  Context:    Indication Delivery
//  Returns:
//  Parameters:
//      handle      An opaque ID returned to the application
//      sender      Address of the node delivering the indication
//      compId      Component with new language mapping
//      language    language-code|territory-code|encoding
*F*/

struct DmiLanguageAddedIN {
    DmiUnsigned_t      handle;
    DmiNodeAddress_t*  sender;
    DmiId_t            compId;
    DmiString_t*       language;
};


/*****************************************************************
 * DmiLanguageDeleted
 *****************************************************************/

/*F*
//  Name:       DmiLanguageDeleted
//  Purpose:    A component language mapping was deleted
//  Context:    Indication Delivery
//  Returns:
//  Parameters:
//      handle      An opaque ID returned to the application
//      sender      Address of the node delivering the indication
//      compId      Component with deleted language mapping
//      language    language-code|territory-code|encoding
*F*/

struct DmiLanguageDeletedIN {
    DmiUnsigned_t      handle;
    DmiNodeAddress_t*  sender;
    DmiId_t            compId;
    DmiString_t*       language;
};


/*****************************************************************
 * DmiGroupAdded
 *****************************************************************/

/*F*
//  Name:       DmiGroupAdded
//  Purpose:    A group was added to a component
//  Context:    Indication Delivery
//  Returns:
//  Parameters:
//      handle    An opaque ID returned to the application
//      sender    Address of the node delivering the indication
//      compId    Component with new group added
//      info      Information about the group added
*F*/

struct DmiGroupAddedIN {
    DmiUnsigned_t      handle;
    DmiNodeAddress_t*  sender;
    DmiId_t            compId;
    DmiGroupInfo_t*    info;
};
```

```
/*****************************************************************
 * DmiGroupDeleted
 *****************************************************************/

/*F*
//  Name:       DmiGroupDeleted
//  Purpose:    A group was deleted from a component
//  Context:    Indication Delivery
//  Returns:
//  Parameters:
//      handle     An opaque ID returned to the application
//      sender     Address of the node delivering the indication
//      compId     Component with the group deleted
//      groupId    Group deleted from the component
*F*/

struct DmiGroupDeletedIN {
    DmiUnsigned_t      handle;
    DmiNodeAddress_t*  sender;
    DmiId_t            compId;
    DmiId_t            groupId;
};


/*****************************************************************
 * DmiSubscriptionNotice
 *****************************************************************/

/*F*
//  Name:       DmiSubscriptionNotice
//  Purpose:    Information about an indication subscription
//  Context:    Indication Delivery
//  Returns:
//  Parameters:
//      handle     An opaque ID returned to the application
//      expired    True=expired; False=expiration pending
//      rowData    Row information to identify the subscription
*F*/

struct DmiSubscriptionNoticeIN {
    DmiUnsigned_t      handle;
    DmiNodeAddress_t*  sender;
    DmiBoolean_t       expired;
    DmiRowData_t       rowData;
};


program DMI2_CLIENT {
     version RMI_CLIENT_VERSION {
        DmiErrorStatus_t _DmiDeliverEvent( DmiDeliverEventIN ) = 0x100;
        DmiErrorStatus_t _DmiComponentAdded( DmiComponentAddedIN ) = 0x101;
        DmiErrorStatus_t _DmiComponentDeleted( DmiComponentDeletedIN ) = 0x102;
        DmiErrorStatus_t _DmiLanguageAdded( DmiLanguageAddedIN ) = 0x103;
        DmiErrorStatus_t _DmiLanguageDeleted( DmiLanguageDeletedIN ) = 0x104;
        DmiErrorStatus_t _DmiGroupAdded( DmiGroupAddedIN ) = 0x105;
        DmiErrorStatus_t _DmiGroupDeleted( DmiGroupDeletedIN ) = 0x106;
        DmiErrorStatus_t _DmiSubscriptionNotice( DmiSubscriptionNoticeIN ) = 0x107;
     } = 0x1;
} = 0x20000000;
```

APPENDIX D - RELATED DOCUMENTS

PC Systems Standard MIF Definition
Release Version 1.1.3
PC Systems Working Committee
27 March 1995

Software Standard Groups Definition
Version 2.0
Software Working Committee
29 November 1995

International Standard
ISO 8859-1
Information processing — 8 bit single-byte coded graphic character set

Desktop Management Interface (DMI) Compliance Guidelines
Version 1.1
September 1995
Steering Committee

Distributed Management Task Force: Enabling your product for manageability with MIF files.
Version 1.1
November 1994
Technical Committee

Distributed Management Task Force: Contacting the DMTF
Version 1.1
November 1994
Steering Committee

LAN Adapter Standard Groups Definition
Version 1.1
April 1994
LAN Adapter Working Group (WG-NIC)

Monitor Standard Groups Definition
Version 1.1
January 1996
Technical Committee

Printer Standard MIF
Version 1.1
Printer Working Group

Finisher Standard MIF
Version 1.1
Large Mailroom Operation Working Group

Systems Standard Groups Definition
Version 1.1
January 1996
Server Working Group

Guide to Writing DCE Applications
2nd Edition, May 1994
John Shirley, Wei Hu, and David Magid
O'Reilly & Associates, Inc.

Distributing Applications Across DCE and Windows NT
1st Edition, November, 1993
Ward Rosenberry and Jim Teague
O'Reilly & Associates, Inc.

Microsoft RPC Programming Guide
March, 1995
John Shirley and Ward Rosenberry
O'Reilly & Associates, Inc.

DCE Security Programming
1st Edition, July 1995
Wei Hu
O'Reilly & Associates, Inc.

Open Software Foundation
World Wide Web Homepage
http://www.osf.org

Power Programming with RPC
John Bloomer
O'Reilly & Associates Inc
1-800-338-6887 US/Canada

International Standard
ISO 10646 Unicode

Desktop Management Interface Specification
Version 1.1
April 1994
Desktop Management Task Force

Secure DMI Overview
DMI Security Working Committee
December 1997
Desktop Management Task Force

Network Security
Kaufman, Perlman, Speciner
1995
Prentice-Hall

NetWare Software Developer's Kit
Novell

Applied Cryptography
Bruce Schneier
1996
Wiley

DMI 2.0, Errata #1
August 6, 1997
Distributed Management Task Force

APPENDIX E - GLOSSARY

Authentication	The process of reliably verifying the identity of a communicating party. For example, a login process is an authentication of a user by an operating system.
Authorization	The process by which a provider decides whether to honor a request or not (usually according to the authenticated identity of the requesting party and the policy). For example, a file system may check the permission list associated with each file in order to authorize a user to access a file. This permission list maps between file operations (like read or write) and user groups.
Attribute	A piece of information about a *component*.
Class string	A text string that identifies a *group* outside the context of a particular *component* declaration. Identical group definitions will have identical class strings.
CMIP	Common Management Information Protocol, an OSI-based network management protocol standardized by ISO.
Command Block	The concatenation of data blocks (data structures) that constitute a command to be sent between *management applications* and the *service provider* and between the Service provider and *component instrumentation*.
Component	Any hardware, software or firmware element contained in (or primarily attached to) a computer system.
Component Instrumentation	The executable code that provides *DMI* management functionality for a particular *component*.
Component Interface (CI)	The *DMI* layer used by *component instrumentations*.
Confirm	The final response from a *Request*.
Confirm Buffer	The area of memory where a *component instrumentation* or *service provider* puts response data.
Credentials	A set of parameters uniquely identifying a principal in the system. The credentials may also contain authentication-related parameters (such as password hash or trusted certificate authority signature).
Direct Interface	Method by which a *component instrumentation* informs the *service provider* that it (the component instrumentation) is already running. Rather than starting the code to service incoming requests, the service provider will use the already running code.
DMI	Desktop Management Interface, the subject of this specification.
DMI Security Indications	Special type of ***DMI*** indications generated by a ***DMIv2.0s*** Service Provider upon performing certain ***DMI*** requests.
DMTF	Distributed Management Task Force
Event	A type of *indication* (unsolicited report) that originates from a *component instrumentation.*
Event Generator	A hardware or software device that has undergone a change in state or in which a certain condition of interest has occurred. This change of state or condition will directly or indirectly cause a new event to be processed by the service provider which then produces and delivers an Indication data structure to event consumers that have registered their interest in receiving Indications.

Event Reporter	The software entity that causes a new DMI event to be processed by the service provider. Events are "reported" by calling the service provider entry point *DmiIndicate()*.
Event Consumer	A software entity that has registered with the service provider through the MI with a non null indication callback procedure address.
Group	A collection of *attributes*. A group with multiple instances is called a *table*.
Indication	An unsolicited report, either from a *component instrumentation* to the *service provider*, or from the service provider to a *management application*.
Inpersonation	The process of faking the identity of a principal in order to receive authorization. Authentication should prevent this security violation.
Integrity	A property of a communication protocol that ensures that data received has not been modified by an unauthorized principal and is identical to the data that was transmitted. Integrity mechanisms can be based on a checksum computed on the transmitted message; messages received with an incorrect checksum are discarded.
ISO 8859-1	A character encoding standard defined by ISO. Commonly known as extended ASCII or 8-bit ASCII.
Kerberos	An authentication system developed at MIT.
Key	An identifier of a particular instance (row) of a table.
Local Interface	A ***DMI*** interface that can be accessed within the managed system, usually through a well known entry point in a DLL or a system call. Note that remote procedure calls from the managed system to itself are not considered a local interface, and RPC mechanisms apply.
Localized String	A version of a display string that is a translation of the original string into an equivalent string in the appropriate local language.
Logging	The process of keeping a record of events that might have some security significance, such as when access to resources has occurred.
Management Agent	A network management protocol agent (such as SNMP or CMOL) that can communicate to the DMI through the MI.
Management Application	Code that uses the MI to request management activity from components.
Management Interface (MI)	The DMI layer between management applications and the service provider.
MIF	Management Information Format; the format used by the DMI for describing components.
MIF Database	The collection of known *MIF files*, stored by the *service provider* (in an implementation-specific format) for fast access.
MIF File	A file that uses the *MIF* to describe a *component*.
Octet	An 8-bit quantity.
One time authentication	The authentication process is done only once in an active session between two parties, usually at the beginning of the session.
Policy	A set of rules that define the actions that various entities can perform on an object based on their identity. For example, the access control list of a file represents the policy for accessing the file including which users have read and write access to the file.
Principal	A completely generic term used by the security community to include both people and computer systems. A principal uniquely represents a security 'object' or 'thing' or 'person'.

Privacy	A property of a communication protocol that ensures that the data exchanged can be disclosed only by its intended recipient; that is, the data will remain opaque for any unauthorized party trying to decode it.
Privileged user	A special user identified by the system as having operating system administration rights, such as an OS administrator or OS backup operator.
Request	A command with associated context issued from the *management application* to accomplish management.
Response	The final response from an *Indication*.
Role	A logical entity that has a name and a set of authorization permissions. Usually there is a set of principals associated with a role.
Row	An instance of a *table*.
Service provider (SL)	The code between the MI and CI that arbitrates access to *component instrumentation* and manages the *MIF database*.
SNMP	Simple Network Management Protocol, an Internet-based network management protocol standardized by the IETF.
System	A computer.
Table	A multidimensional *group*; a group with more than one instance.
Ticket	A data structure constructed by a trusted intermediary to enable two parties to authenticate.
Transport	The 4th Layer in the 7-Layer OSI networking model. IP is an example of a common network transport.
Unicode	A character encoding standard defined by the Unicode Consortium. Unicode characters are 2 octets each. When the first octet is zero, the second octet maps to the characters in ISO 8859-1.
User	A uniquely-identified principal person user in a multi-user system. A user is represented by its credentials (see *Credentials*).
X.509	A CCITT standard for security services within the X.500 directory services framework. The X.509 encoding of public key certificates has been widely adopted.

INDEX