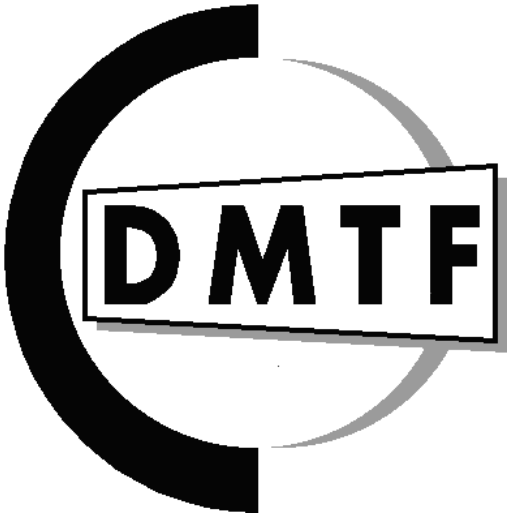
Desktop Management Task Force



Desktop Management Interface Specification

Version 1.0

29 April 1994

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

Desktop Management Task Force (DMTF)
M/S JF2-51
2111 N.E. 25th Avenue
Hillsboro, OR 97124
FAX: (503) 696-9027
email: dmtf-info@dmtf.org

Additional copies of this specification may be obtained free of charge electronically via internet at:
ftp.dmtf.org
or
hardcopy for a fee by contacting:

Desktop Management Task Force
M/S JF2-51
2111 N.E. 25th Avenue
Hillsboro, OR 97124
PHONE: (503) 696-9300
FAX: (503) 696-9027

IMPORTANT INFORMATION AND DISCLAIMERS

1. THIS SPECIFICATION (WHICH SHALL INCORPORATE ANY REVISIONS, UPDATES, AND MODIFICATIONS HERETO) IS FURNISHED FOR INFORMATIONAL PURPOSES ONLY. INTEL CORPORATION, MICROSOFT CORPORATION, DIGITAL EQUIPMENT CORPORATION, HEWLETT-PACKARD, INTERNATIONAL BUSINESS MACHINES CORPORATION, NOVELL INC., SUNCONNECT (A SUN MICROSYSTEMS COMPANY), SYNOPTICS COMMUNICATIONS, INC. OR ANY OTHER DMTF MEMBER MAKE NO WARRANTIES WITH REGARD THERETO, AND IN PARTICULAR DO NOT WARRANT OR REPRESENT THAT THIS SPECIFICATION OR ANY PRODUCTS MADE IN CONFORMANCE WITH IT WILL WORK IN THE INTENDED MANNER OR BE COMPATIBLE WITH OTHER PRODUCTS IN NETWORK SYSTEMS. NOR DO THEY ASSUME RESPONSIBILITY FOR ANY ERRORS THAT THE SPECIFICATION MAY CONTAIN OR HAVE ANY LIABILITIES OR OBLIGATIONS FOR DAMAGES INCLUDING, BUT NOT LIMITED TO, SPECIAL, INCIDENTAL, INDIRECT, PUNITIVE, OR CONSEQUENTIAL DAMAGES WHETHER ARISING FROM OR IN CONNECTION WITH THE USE OF THIS SPECIFICATION IN ANY WAY. CORPORATIONS MAY FOLLOW OR DEVIATE FROM THIS SPECIFICATION AT ANY TIME.

2. NO REPRESENTATIONS OR WARRANTIES ARE MADE THAT ANY PRODUCT BASED IN WHOLE OR IN PART ON THE ABOVE SPECIFICATION WILL BE FREE FROM DEFECTS OR SAFE FOR USE FOR ITS INTENDED PURPOSE. ANY PERSON MAKING, USING OR SELLING SUCH PRODUCT DOES SO AT HIS OWN RISK.

3. THE USER OF THIS SPECIFICATION HEREBY EXPRESSLY ACKNOWLEDGES THAT THE SPECIFICATION IS PROVIDED AS IS, AND THAT THE DMTF, NEITHER INDIVIDUALLY NOR COLLECTIVELY, MAKE ANY REPRESENTATIONS, EXTEND ANY WARRANTIES OF ANY KIND, EITHER EXPRESS OR IMPLIED, ORAL OR WRITTEN, INCLUDING ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, OR WARRANTY OR REPRESENTATION THAT THE SPECIFICATION OR ANY PRODUCT OR TECHNOLOGY UTILIZING ANY ASPECT OF THE SPECIFICATION WILL BE FREE FROM ANY CLAIMS OF INFRINGEMENT OF INTELLECTUAL PROPERTY, INCLUDING PATENTS, COPYRIGHT AND TRADE SECRETS OF ANY THIRD PARTY, OR ASSUMES ANY OTHER RESPONSIBILITIES WHATSOEVER WITH RESPECT TO THE SPECIFICATION OR SUCH PRODUCTS. IN NO EVENT WILL DMTF MEMBERS BE LIABLE FOR ANY LOSSES, DAMAGES INCLUDING, WITHOUT LIMITATION, THOSE DAMAGES DESCRIBED IN SECTION 1 ABOVE, COSTS, JUDGMENTS, OR EXPENSES ARISING FROM THE USE OR LICENSING OF THE SPECIFICATION HEREUNDER.

Contents

Contents 2

1. Introduction 4
   1.1. Motivation 4
   1.2. Basic terminology 4
   1.3. Overview 4
   1.4. Data model 5
   1.5. Service primitives 6
   1.6. Sample flow diagrams 7
      A request to two component instrumentations 7
      [1]An event from a component instrumentation 8
   1.7. Operational characteristics 9
   1.8. Commands 9
   1.9. Operational details 10

2. The Service Layer 12
   2.1. Entry points 12
   2.2. Responsibilities 12

3. Management Information Format 14
   3.1. Lexical conventions 14
   3.2. Comments 14
   3.3. Keywords 15
   3.4. Data types 15
   3.5. Constants 16
   3.6. Block scope 17
   3.7. Language statement 17
   3.8. Common statements 18
      3.8.1. Name statement 18
      3.8.2. Description statement 18
      3.8.3. ID statement 19
   3.9. Component definition 19
   3.10. Path definition 19
   3.11. Enum definition 20
   3.12. Group definition 20
      3.12.1. Class statement 21
      3.12.2. Key statement 21
   3.13. Attribute definition 22
      3.13.1. Type statement 22
      3.13.2. Access statement 22
      3.13.3. Storage statement 23
      3.13.4. Value statement 23
   3.14. Group example 23
   3.15. Populating tables 24
   3.16. The ComponentID group 25

[1]

4. Common Data Blocks 26
4.1. DmiMgmtCommand 26
4.2. DmiCiCommand 28
4.3. DmiConfirm 28
4.4. DmiIndicate 29
4.5. DmiVersion 30
4.6. DmiGroupKeyData 30
4.7. DmiTimeStamp 30

5. The Management Interface 32
5.1. Invoking a command 32
5.2. Canceling 32
5.3. Registration 33
5.3.1. Unregistering 34
5.4. Indications 34
5.5. List commands 34
5.5.1. Listing components 34
5.5.2. Listing groups 37
5.5.3. Listing attributes 38
5.5.4. Listing descriptions 40
5.6. Getting attributes 41
5.7. Setting attributes 42
5.7.1. Reserving attributes 43
5.7.2. Releasing attributes 44
5.8. Getting rows 44
Note for component providers 45
5.9. Implementation-specific commands 45

6. The Component Interface 47
6.1. Installation 47
6.2. Removal 48
6.3. Invoking a command 49
6.4. Canceling 49
6.5. Registration 49
6.6. Sending events 51

7. Compliance 53

8. Command and Status Codes 54
8.1. Command codes 54
8.2. Status codes 56

9. MIF Grammar 58

A. ISO 639 63

B. ISO 3166 64

C. Sample MIF 66

D. Glossary 71

E. Index 73

1. Introduction

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
- mappable to existing management protocols (e.g., CMIP, SNMP)

The DMI does not address or specify a protocol for management over a network.

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.

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

1.3. Overview

The DMI has four elements:

1. a format for describing information
2. a format for data transfer
3. a mechanism for data transfer
4. a set of services for facilitating 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***.

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.

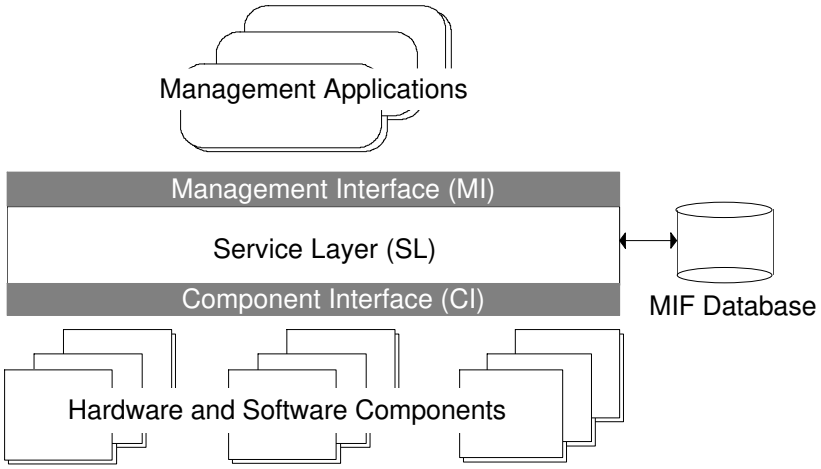
The ***Management Interface***, or ***MI***, is used by applications that wish to manage components. The MI shields management application vendors from the different mechanisms used to obtain management information from elements within a computer system.

The CI and MI are data interfaces, as opposed to procedural interfaces. Data blocks describe the format for data transfer, not parameters to a function call. Once the set of blocks that comprise a command are built, a single function call is used to move the blocks through the DMI.

The ***Service Layer***, or ***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.

The DMI is a local interface, to be used within a single system. The DMI is not designed to replace existing network management protocols. It is designed to provide a consistent method for providing instrumentation to those protocols. The Service Layer is the broker of local instrumentation.

A functional block diagram is shown below.

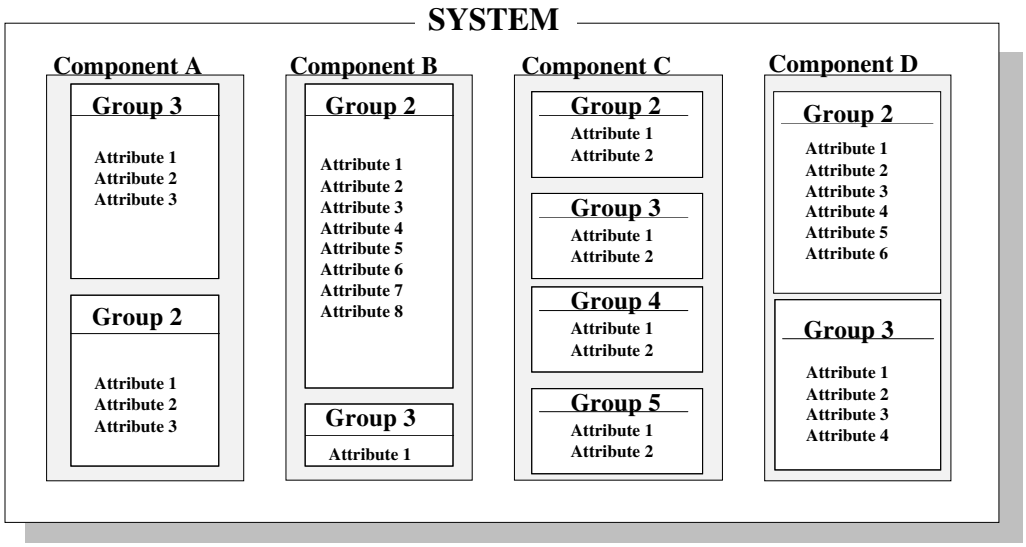


In this diagram, all elements exist within a single system, or are directly attached, such as printers or modems. The management applications may be local command-line or graphical user interface programs, but they may also be network protocol agents that translate between a particular management protocol and the DMI.

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 replicated, such as the set of attributes for each instance of a network interface table. Replicated 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 replicated as a table. The component instrumentation presents this component/group/key/attribute representation to the management application. A diagram is shown below.



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

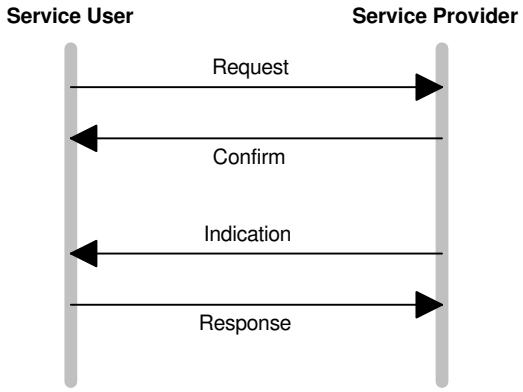
1.5. Service primitives

The DMI interfaces are defined by the primitives that pass through the Service Layer. The Service Layer takes on the role of component instrumentation to all management applications, and the role of a management application to all component instrumentation. This idea is the key to understanding the flow of service primitives through the DMI.

Because the Service Layer takes on both management application and component instrumentation roles, the DMI uses two terms to simplify discussion. A service ***user*** is an entity acting as a management application. A service ***provider*** is an entity acting as a component instrumentation.

Within this context, there are four types of service primitives:

REQUEST	Flows from the user to the provider to initiate a command.
CONFIRM	Flows from the provider to the user in response to a REQUEST.
INDICATION	Flows from the provider to the user to indicate some unsolicited occurrence.
RESPONSE	Flows from the user to the provider in response to an INDICATE.

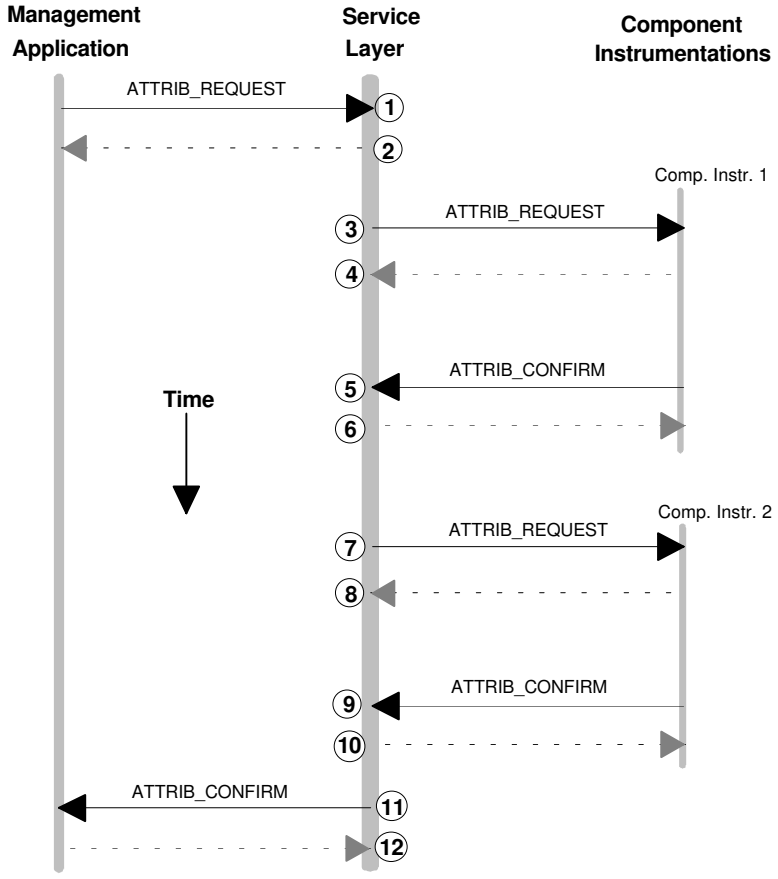


The following section contains examples to illustrate.

1.6. Sample flow diagrams

A request to two component instrumentations

Management applications request information through the Service Layer for a particular component. The Service Layer acknowledges receipt of the message (a synchronous transaction), then issues as many requests to different component instrumentations as necessary to satisfy the original request. In this example, two component instrumentations must be questioned. The Service Layer gets asynchronous confirms back from the component instrumentations. Once the Service Layer has all the information needed to satisfy the request, or an error message from any component instrumentation, a confirm message is asynchronously returned.



The numbers in the above figure indicate the sequence of steps taken:

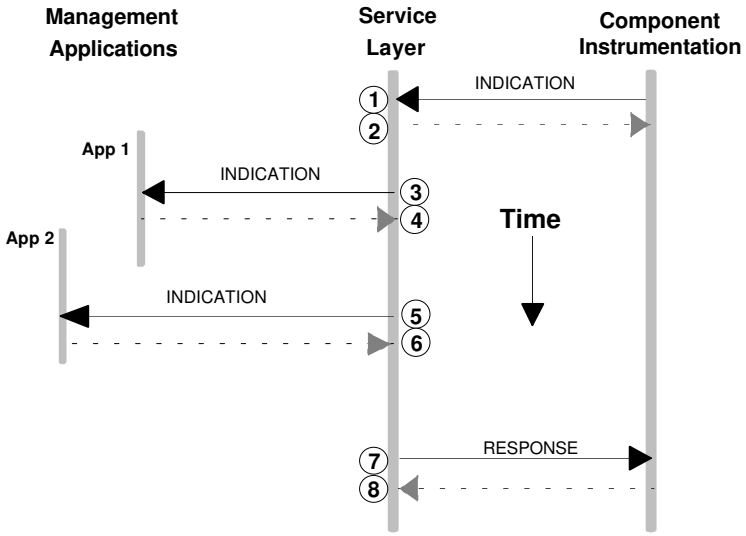
1 The management application issues a REQUEST to the SL.
2 The SL returns from the REQUEST call.
3 The Service Layer issues a REQUEST to component instrumentation 1.
4 Component instrumentation 1 returns from the REQUEST, and begins processing.
5 After completing the REQUEST, component instrumentation 1 issues a CONFIRM to indicate that it has completed the request, and to return the information to the SL.
6 The SL returns from the CONFIRM call.
7 The SL issues a REQUEST to component instrumentation 2.
8 Component instrumentation 2 returns from the REQUEST, and begins processing.
9 After completing the REQUEST, component instrumentation 2 issues a CONFIRM to indicate that it has completed the request, and to return the information to the SL.
10 The SL returns from the CONFIRM call.

11 The SL issues a CONFIRM call to the application to indicate that it has completed the requested action, and to return the results of that operation.
12 The application returns from the CONFIRM call.

In this example the Service Layer waits for a confirm from the first component instrumentation before issuing the second request. In practice it may issue many simultaneous requests, depending on the nature of the original request and the Service Layer implementation.

An event from a component instrumentation

The component instrumentation offers unsolicited information to the Service Layer. Unsolicited information does not mean *unwanted*, only *unexpected*. In the DMI, any unsolicited report is known as an ***indication*** or ***event***[2]. The Service Layer acknowledges receipt of the message (a synchronous transaction), then issues indications to each currently running management application. Management applications usually do not respond to the Service Layer, although Service Layer implementations may require response messages from management applications. Once the Service Layer has notified each registered management application, it responds to the component instrumentation. Between the time the Service Layer returns from the indication and sends a response, the component instrumentation is free to perform other tasks.



The numbers in the above figure indicate the sequence of steps taken:

1 A component instrumentation notifies the SL by issuing an INDICATION.
2 The SL returns from the INDICATION call.
3 The SL notifies management app. 1 of the event by issuing an INDICATION.
4 Management app. 1 returns from the INDICATION call.
5 The SL notifies management app. 2 of the event by issuing an INDICATION.
6 Management app. 2 returns from the INDICATION call.
7 The SL issues a RESPONSE to the component instrumentation.
8 The component instrumentation returns from the RESPONSE call.

The Service Layer repeats steps 3 and 4 for each management application currently registered to receive indications. Once all have been notified, the Service Layer sends a response to the component instrumentation.

[2]This specification uses both terms interchangeably.

1.7. Operational characteristics

The relationship among management applications, the Service Layer and component instrumentation can exist as a many-to-one-to-many relationship. There may be many management applications issuing commands through a single Service Layer to manage many components.

For purposes of identification, management applications must register with the Service Layer before they can participate in management functions. Component instrumentations must install into the Service Layer once when first introduced to the system. They must register with the Service Layer when they wish to provide unsolicited information or to notify the Service Layer of their immediate availability. The mechanics of "connecting" to the Service Layer to register or issue commands will differ among operating systems and SL implementations.

Control flow is usually initiated from the management application to the Service Layer and on to the component instrumentation. The exceptions are *indications*, unsolicited reports that flow in the opposite direction.

The Service Layer returns a status to the management application indicating a request has been successfully queued for processing, or there is some form of error. Any error at this time is fatal. The caller must fix the problem, if possible, and reissue the request.

The confirm message occurs some time later as an asynchronous notification from the Service Layer to the management application. This is made when the request completes.

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 Service Layer gets MIF information from the 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, cancel outstanding commands, and allow component instrumentations to generate indications.

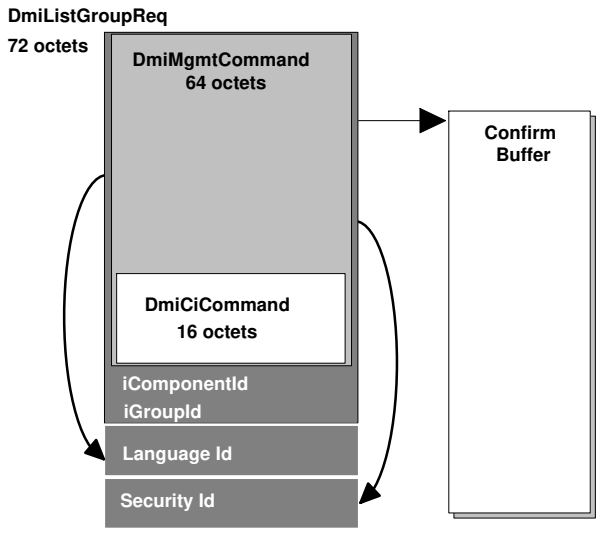
1.8. Commands

Commands are composed of data structures called ***command blocks***. Since commands fall into two classes, requests and indications, command blocks are referred to as either ***request blocks*** or ***indication blocks***.

Command blocks usually start with a standard block called DmiMgmtCommand, then append additional command-specific blocks. Chapter 4 describes the standard command blocks, and chapters 5 and 6 describe the command-specific blocks for the MI and CI.

Within the command block, a ***command code*** is provided to tell the recipient what specific operation to perform, and by association, the composition of the remainder of the command block. The entire set of blocks, with the command code and associated data, form the command. The caller then passes this command block to the Service Layer via a function call.

The following diagram illustrates DmiListGroupReq, the pre-defined request block used to get information about a component's group.



DmiListGroupReq is a 72-octet request block that consists of three elements: the standard 64-octet DmiMgmtCommand block, the command-specific 4-octet component ID, and the command-specific 4-octet group ID. (Note that the DmiMgmtCommand itself embeds the 16-octet DmiCiCommand block).

Variable data, such as strings, cannot appear in the pre-defined command block. Nor can they appear as pointers to memory, as this creates performance, security and implementation problems in some environments. Instead, variable data is stored just past the pre-defined command block, and a value in the block specifies the 32-bit unsigned offset location of the data within the entire command block. In the above example, "Language Id" is an offset string found at offset 72. Offsets of zero are used to indicate that no data is provided.

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 (not characters) in the <*data*>. The length does not include the 4 octets that contain the length, so the storage for the string "hello" in ISO 8859-1 format is nine (9) octets long, but the size given in the length field is five (5) octets. Strings are not required to be zero-terminated as in the C programming language.

A true pointer to memory is required for the confirm buffer. Within the DmiMgmtCommand structure, the confirm buffer is always a pointer, not an offset. Whenever a buffer is specified, a corresponding buffer size is specified.

A variable named iCmdLen within the DmiMgmtCommand block indicates the total length of the command, including offset data. In the example block above, if the "Language Id" is 9 octets -- four octets for string length and five octets for text -- and the "Security Id" was zero (not specified) the total length of the command would be 81 octets. Buffer sizes such as the confirm buffer are not counted as part of the command block length.

1.9. Operational details

The Service Layer guarantees that a successful *single* request for multiple pieces of information will result in a single confirm message with the information arranged in the same order as the request. In other words, if a management application requests A, B and C in that order within a request, the result from the Service Layer will contain A, B and C, in that order.

If the Service Layer cannot perform the operation on all requested attributes, it will return a partial response, rather than failing the entire request. The Service Layer will always abandon processing after the first error. In the above example for A, B and C, if all attributes are managed by a single component instrumentation and a failure occurs while getting B, only A will be returned -- the operation on C will not be attempted. If A, B and C were handled by different component instrumentation code and the Service Layer issued simultaneous requests to all three, it may receive valid data for A and C before getting an error message for B, due to the asynchronous nature of the DMI. In that case, the Service Layer will return only A.

On the other hand, when a management application sends *multiple* requests for single pieces of information, the Service Layer may return whatever responses it gets in any order. The DMI is asynchronous *across* calls, but sequencing is preserved *within* calls.

Operations may only be interrupted by requests to cancel the currently running command. Component instrumentations 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 Service Layer, are the responsibility of the management application. Likewise, any desire for mutual exclusion, to lockout certain accesses, or to provide security in any form, is the responsibility of the management application.

Unless specifically stated otherwise, all fields within a block that reference data use 32-bit unsigned offsets within the buffer, as opposed to pointers. These offsets are the number of octets from the start of the buffer. Throughout this document, the pseudo-type OFFSET is used to refer to this situation. When pointers are specified, the pseudo-type PTR is used.

It is illegal for the confirm buffer to contain pointers into component instrumentation space. In other words, the component instrumentation does not provide the storage space for the confirm buffer. All data must be copied from the component instrumentation space to the confirm buffer.

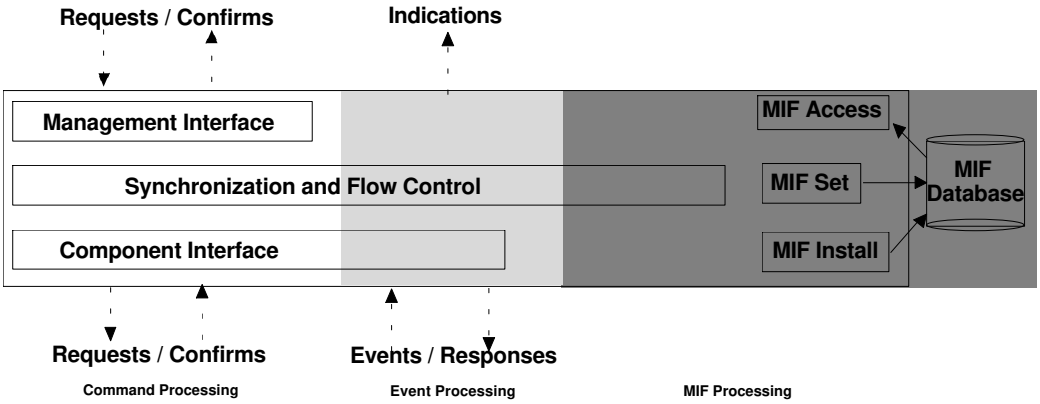
A DMI implementation is specific to its operating environment; however, the layout of the data blocks used to transfer information is not specific to an operating system. Structure alignment is implementation-dependent; since alignment is not guaranteed, items placed in the confirm buffer must be copied into an auxiliary buffer before manipulation, so that problems caused by possible misalignment can be avoided.

2. The Service Layer

The Service Layer (SL) coordinates and arbitrates requests from management applications to the specified component instrumentations. The Service Layer 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 Service Layer or passed directly to the CI. The Service Layer conceptually operates on memory passed from either the management applications or, sometimes in the case of events, the component instrumentation.

The following illustration depicts a possible Service Layer block diagram. This is an example only and is not part of the DMI specification.



2.1. Entry points

As previously mentioned, components and applications communicate with each other through the Service Layer, and communicate with the Service Layer through pre-defined blocks of memory. The Service Layer provides three (3) named function call entry points so the filled-out blocks can be sent to the Service Layer. They are:

Entry Point	*Used By*	*Description*
DmiInvoke()	Management applications Component Instrumentations	Issue Commands, Registration Installation, Registration
DmiCancel()	Management applications	Cancel Commands
DmiIndicate()	Component Instrumentations	Issue Unsolicited Reports

The format and use of these entry points will be discussed later in this specification.

2.2. Responsibilities

The Service Layer must coordinate the dynamic installation and removal of component instrumentations and management applications. It must enforce that at least group 1 (the component ID group) is in each MIF file.

The Service Layer must coordinate the registration of entities wishing to initiate management activities.

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

The Service Layer is responsible for launching the component instrumentation, if necessary, and passing along the command blocks.

When a management application requests more than one attribute value from a component in a single command, the Service Layer sends commands to the component instrumentation for each attribute. This must be done because a command may require alternate access to component instrumentation and to the MIF database. Since the DMI ensures that the ordering of attributes in a confirm buffer is the same as the ordering of attributes in the original request buffer, this "command slicing" prevents Service Layer buffer management problems. It also makes writing component instrumentation code slightly easier.

The Service Layer 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 Service Layer must forward indications from component instrumentation to each registered management application, and must time-stamp incoming indications before forwarding them.

The Service Layer must send indications to all registered management applications when components are installed or removed from the MIF database.

The Service Layer must appear to management applications as a component with ID 1 (one). As a component, it must support the standard ComponentID group (defined in the next chapter). Also like a component, it may define additional groups beyond the ComponentID group.

3. 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 Service Layer 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 Chapter 9. A sample MIF file is given in Chapter 10.

3.1. 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 Service Layer 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, newlines, 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.

3.2. 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.

3.3. 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

3.4. Data types

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

Data 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
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 value *n* represents the maximum number of octets in the string. The actual number of octets in use may be shorter than this maximum value. Strings are stored with their length in the first four octets (which are not counted in *n*); they do not need to be zero-terminated as in the C/C++ programming languages. String lengths represent the number of octets in the string, not the number of characters.

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.

3.5. Constants

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

Syntax	*Base*
nnn	decimal
0*nnn*	octal
0x*nnn* or 0X*nnn*	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
\x*hh*	bit pattern, hexadecimal
ooo	bit pattern, octal

If the character following a backslash is not one those specified, the backslash is ignored.

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.

3.6. 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
```

3.7. Language statement

The **language** statement is used to describe the native (human) language of the MIF file. This optional statement, if provided, 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.

Since users may edit the MIF file to translate the literal strings to a different language, they may change the language string also. The language statement may appear only once per MIF file.

Samples of the codes defined in the two ISO standards are in Appendices A and B.

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. Do not localize class strings or language names. Keywords must not be localized.

3.8. 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.

3.8.1. 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.

3.8.2. 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.

3.8.3. 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 Service Layer at installation time. The id statement may appear only once per definition.

3.9. Component definition

The component definition has the following syntax:

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

Only one component definition may appear in a MIF file.

3.10. Path definition

Path definitions are used to relocate 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** and **win32**. Case is not significant.

If the component instrumentation is provided by code that will connect to the Service Layer (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.dll"
    os2   = direct-interface
    dos   = "C:\\someplace\\doscode.com"
    unix  = "/someplace/unixcode"
end path
```

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

See the sample MIF for usage of the symbols defined in the path definition.

3.11. 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.

3.12. 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]...]

          ( 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 managed by the component instrumentation code. Any subsequent **table** definitions (defined below) within the component definition cannot use this group definition as a template.

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 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.

3.12.1. 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.

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 ("1", "A", "FIRST ONE EVER", etc.).

Essentially the class string is an opaque string, and any convention may be used. However, since applications and Service Layers 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|1.0" is not the same as "DMTF | Sample | 1.0" 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 may simply omit the definitions for the unsupported attributes. A better method is to use the **unsupported** keyword within the attribute definition (defined below). Management applications must be sufficiently robust to deal with subsets of a class.

Only one class statement is allowed per group.

3.12.2. 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 appli-

cations send requests or component instrumentations 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.

3.13. 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]
          [storage = storagetype]
          [value = [v | * "name" | "enum string"]]
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.

3.13.1. 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 3.4.

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.

3.13.2. 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.

3.13.3. Storage statement

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

```
storage = where
```

where *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.

3.13.4. Value statement

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

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

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 Service Layer 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 Service Layer 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. It is illegal to specify an enumeration value for write-only attributes. Note that specifying an integer for an enumeration is acceptable.

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 Service Layer that this attribute is not supported by this component.

The value statement is required except when defining table templates, when 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.

3.14. Group example

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

```
Start Group
        Name  = "Software Template"
        Class = "DMTF|Software Example|1"
        Key   = 1                          // key on Product Name
        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.

3.15. 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.

Within a table row, the values are provided as in section 3.13.4, 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 3.14:

```
Start Table
        Name        = "Software Table"
        Class       = "DMTF|Software Example|1"
        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. Service Layers 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.

3.16. 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** as its value.

The ComponentID class string is "**DMTF|ComponentID|1.0**". The attributes in the group are all read-only. Their definitions are:

Name	*ID*	*Type*	*Storage*	*Description*
"Manufacturer"	1	string (64)	common	The organization that produced this component
"Product"	2	string(64)	common	The name of this component or product
"Version"	3	string(64)	specific	The version string for this component
"Serial Number"	4	string(64)	specific	The serial number for this component
"Installation"	5	date	specific	The time and date of the last install of the component on this system
"Verify"	6	integer	common	The verification level for this component

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	*Meaning*
0	an error occurred; check status code
1	component does not exist
2	verify not supported
4	component exists, functionality untested
5	component exists, functionality unknown
6	component exists, functionality no good
7	component exists, functionality good

4. Common Data Blocks

The DMI uses a common set of data blocks. Management applications allocate request and confirm blocks and pass them to the Service Layer. The Service Layer uses the same (or similar) data blocks to call the component instrumentation through the CI. Likewise, the component instrumentation allocates indication and response blocks and passes them to the Service Layer, which uses the same (or similar) data blocks to call the management application through the MI.

This section describes the commonly used data blocks. Throughout this specification, the tables that describe a block have two columns, Offset and Size, that describe the relative start of the element and its size, both in octets.

4.1. DmiMgmtCommand

MI commands start with a DmiMgmtCommand block. Each outstanding command requires a separate DmiMgmtCommand block. Commands will usually consist of this block with additional parameters appended to this block.

Offset	*Size*	*Type*	*Variable Name*
0	4	INT	iLevelCheck
4	4	INT	iCommand
8	4	INT	iCmdLen
12	4	INT	iMgmtHandle
16	4	INT	iCmdHandle
20	4	OFFSET	osLanguage
24	4	OFFSET	oSecurity
28	4	INT	iCnfBufLen
32	4	PTR	pCnfBuf
36	4	INT	iRequestCount
40	4	INT	iCnfCount
44	4	INT	iStatus
48	16	STRUCT	DmiCiCommand

iLevelCheck a 4-octet value with the value: 0x444d4931. In ISO 8859-1 this is "DMI1". This value performs two functions: first, as a check so the receiver can be confident that the pointer is pointing to a valid command block. Second, this value indicates the revision level of the DMI specification used by the code issuing the command. Future versions of the DMI specification will use an incremental numbering scheme, so the next revision will be 0x444d4932, or "DMI2".

iCommand a code specifying the command to execute. The standard command codes are listed in chapter 8 as well as with the documentation of each command. Service Layers may optionally accept non-standard command codes; these commands are defined by the SL implementation and exist in the range above **0x80000000** (the high bit of the command is set).

iCmdLen length in octets of the entire command, including the DmiMgmtCommand block, all appended blocks, and any values stored at the end of the buffer for Set Attribute commands.

iMgmtHandle a value that uniquely identifies the management application. The management application originally obtained this value from the DmiRegisterMgmtCmd command. This value is ignored when sending the DmiRegisterMgmtCmd command.

iCmdHandle optional integer set by the management application to a value unique to all other outstanding commands from that management application. The handle can later be used to cancel the command.

osLanguage offset to a string indicating the desired (human) language for the response. See section 3.7 for the format of the string. If the management application sets this field to zero, a language string of `"en|US|iso8859-1"` is assumed.

oSecurity offset to an optional security credentials block, or zero if no block is specified. Service Layers and component instrumentations may use the information in this block to implement local security policies as needed. The block looks like:

DmiSecurity

Offset	*Size*	*Type*	*Variable Name*
0	4	INT	iSecurityLen
4	4	INT	iSecurityType
8	x	STRUCT	SecurityData

iSecurityLen length of the security block, in octets of contiguous memory.

iSecurityType type of security block being passed in SecurityData. Standard security blocks are not provided in DMI 1.0. Service Layers may optionally accept non-standard security blocks; these blocks are defined by the SL implementation and exist in the range above **0x80000000** (the high bit of the type is set).

SecurityData a block that contains the details of the security credentials. The size and structure of the security data depend upon the value of iSecurityType.

iCnfBufLen length of the confirm buffer, in octets of contiguous memory. The management application sets this to tell the Service Layer the maximum size of the response.

pCnfBuf pointer to the buffer where the management application would like the confirm to go. The management application must supply this buffer; it is not provided by the Service Layer or component instrumentation. The buffer must be a minimum of 512 octets.

iRequestCount the number of requests in this command. Note that this count does not include the DmiMgmtCommand block itself.

iCnfCount (set on return) number of valid confirm blocks in pCnfBuf.

iStatus (set on return) return status of this command. In the case of an error, where iCnfCount contains the number of valid confirms, iStatus refers to the request represented by (*iCnfCount* + 1).

4.2. DmiCiCommand

The DmiCiCommand block is included in the DmiMgmtCommand block for the component instrumentation.

Offset	*Size*	*Type*	*Variable Name*
0	4	OFFSET	oCmdListEntry
4	4	INT	iCnfBufLen
8	4	PTR	pCnfBuf
12	4	PTR	pConfirmFunc

oCmdListEntry offset to the command to be processed. This is only used for Get and Set Attribute commands since there can be multiple commands in the command block. For Get Attribute commands, it will be an offset to the desired DmiGetAttributeData block. For Set Attribute commands, it will be an offset to the desired DmiSetAttributeData block. For other commands, it will be zero.

iCnfBufLen length of the confirm buffer in octets of contiguous space. This is not necessarily the buffer length listed in DmiMgmtCommand -- the Service Layer may have supplied a different buffer just for this command.

pCnfBuf pointer to the confirm buffer supplied by the Service Layer where the result should be placed. This is not necessarily the buffer pointed to by pCnfBuf in the DmiMgmtCommand block.

pConfirmFunc() entry point into the Service Layer where the component instrumentation should call upon completion of the command. The C prototype for this call is:

```
void pConfirmFunc(DmiConfirm *buf)
```

where buf points to a DmiConfirm block, described next.

4.3. DmiConfirm

The DmiConfirm block is sent as the result of a request message, to indicate that processing of the request has completed.

Offset	*Size*	*Type*	*Variable Name*
0	4	INT	iLevelCheck
4	4	PTR	pDmiMgmtCommand
8	4	INT	iStatus

iLevelCheck a 4-octet value with the value: 0x444d4931. In ISO 8859-1 this is "DMI1". This value serves two purposes: first, as a check so the receiver can be confident that the pointer is pointing to a valid confirmation block. Second, this value indicates the revision level of the DMI specification used by the code issuing the confirm message. Future versions of the DMI specification will use an incremental numbering scheme, so the next revision will be "DMI2".

pDmiMgmtCommand pointer to a command block.

iStatus a status code indicating any error or other information.

4.4. DmiIndicate

Entities within the DMI may issue unsolicited reports. Component instrumentations may issue events to the Service Layer. The Service Layer may issue notifications of component installations and removals to management applications. These unsolicited notifications are called *indications*. Each sender of an indication fills out a DmiIndicate block; this is passed to the receiver's indication entry point that was obtained during the registration process.

Offset	*Size*	*Type*	*Variable Name*
0	4	INT	iLevelCheck
4	4	INT	iIndicationType
8	4	INT	iCmdLen
12	4	INT	iComponentId
16	28	TIMESTAMP	DmiTimeStamp
44	4	PTR	pResponseFunc
48	4	OFFSET	oIndicationData

iLevelCheck a 4-octet value with the value: 0x444d4931. In ISO 8859-1 this is "DMI1". This value performs two purposes: first, as a check so the receiver can be confident that the pointer is pointing to a valid indication block. Second, this value indicates the revision level of the DMI specification used by the code issuing the indication. Future versions of the DMI specification will use an incremental numbering scheme, so the next revision will be "DMI2".

iIndicationType a code specifying the indication type. The possible codes are:

1 Event
2 Install
3 Uninstall
4 MIF Addition

iCmdLen length of the entire indication block, in octets.

iComponentId the ID of the component instrumentation generating this indication. If the Service Layer has initiated the indication, this field is one (1).

DmiTimeStamp a DmiTimeStamp block (described below). If a component instrumentation initiated the indication and set the time to zero, the Service Layer will add the time stamp. If the field is non-zero, the Service Layer will not check the value for accuracy.

pResponseFunc a pointer to the entry point that will be called when the receiver is finished processing the indication report. Normally this is only for use between component instrumentation and the Service Layer; the SL will set this to zero before sending indications to management applications. Some SL implementations may require a response to certain indications, in which case the field will not be zero and the management application must call the SL entry point.

oIndicationData an offset to a required buffer that provides context for the indication. The meaning of this data is dependent on the indication type. For events, the DmiEventData block is returned. For component installs and removals, a DmiListComponentCnf block is returned. Both are described later.

4.5. DmiVersion

This block describes the version of the Service Layer implementation. It is returned by the registration calls, DmiRegisterMgmtCmd and DmiRegisterCiCmd. The block looks like:

Offset	*Size*	*Type*	*Variable Name*
0	4	OFFSET	osDmiSpecLevel
4	4	OFFSET	osImplDesc

osDmiSpecLevel offset to a string that describes the latest level of the DMI specification supported by this Service Layer. Service Layers that support this document will use the string "DMI 1.0".

osImplDesc offset to a string that describes the implementation-specific version of the Service Layer.

4.6. DmiGroupKeyData

This block defines a key to be used when indexing into a Group that is arranged as a table. There must be as many of these blocks as there are integers in the **key** statement within the component definition. For example, if **key** = 4,7,3 then there must be three DmiGroupKeyData blocks, with the first one for attribute 4, followed by attribute 7, then attribute 3. Out-of-order keys are not supported. Partial keys are also not supported — a key must uniquely identify no more than one instance of a table.

Offset	*Size*	*Type*	*Variable Name*
0	4	INT	iAttributeId
4	4	INT	iAttributeType
8	4	OFFSET	oKeyValue

iAttributeId the attribute to be used as an index into a table.

iAttributeType data type (Counter, Gauge, etc.). Legal values are:

0	unknown (usually indicates a MIF database error)
1	counter
2	counter64
3	gauge
5	integer
6	integer64
7	octet string
8	displaystring, string
11	date

oKeyValue offset to the value of the key.

The attribute ID and type are necessary even though the **key** statement determines the order, because the block does triple-duty. It is used by management applications to specify keys; by the Service Layer for returning the group's key structure; and by component instrumentation to return key values. The fields must always be filled, even if they are "not needed."

4.7. DmiTimeStamp

This block describes the time format used in the 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:

Octets	*Contents*	*Encoding*
1-4	year	decimal
5-6	month	decimal (1..12)
7-8	day	decimal (1..31)
9-10	hour	decimal (0..23)
11-12	minutes	decimal (0..59)
13-14	seconds	decimal (0..60)
15	dot	"."
16-21	microseconds	decimal (0..999999)
22-25	offset from UTC (in minutes)	["+" | "-"] (000..720)
26-28	unused	(for alignment)

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 Service Layer is not required to check the contents of this string for validity.

5. The Management Interface

The Management Interface allows a management application to ask for lists of managed components, access specific components, and get and set individual groups and attributes. Additionally, the management application uses this interface to register for event notifications.

A command is issued from the management application to the Service Layer. It consists of a sequence of data blocks starting with the DmiMgmtCommand block followed by a number of blocks of various types based on the nature of the command.

5.1. Invoking a command

Because the DMI is a data interface, all commands are specified with data blocks. However, there is one function call provided so the management application can send the command block to the Service Layer. The ***DmiInvoke***() call sends the command (as defined by the DmiMgmtCommand block and any following blocks) to the Service Layer for processing. The C-language prototype for this call is:

```
unsigned long  DmiInvoke(PTR command)
```

where command is the complete command block. The return result is a 32-bit status value indicating success or failure. The possible status values are described in Chapter 8.

The Service Layer will immediately return control to the management application, while it (the Service Layer) processes the command. While the Service Layer processes the command, the management application is free to continue processing and issue additional commands. It cannot re-use the DmiMgmtCommand buffer it handed to the Service Layer -- simultaneous commands from a management application must use different DmiMgmtCommand blocks. When the Service Layer is done with the command, it notifies the management application by calling the function pConfirmFunc() that was given when the management application originally registered with the Service Layer. At that point, the management application can re-use the DmiMgmtCommand block.

Note that in some operating environments, it is theoretically possible to get a confirm message before DmiInvoke() returns, due to the asynchronous nature of the DMI. Service Layer implementations in such environments should document the possible behavior.

The DMI specification does not include a time-out policy. If a running command appears (to the requester) to be not responding, the management application should issue a cancel command.

5.2. Canceling

A management application may choose to cancel an outstanding command. The ***DmiMgmtCommand*** block can be used in these cases. In commands where actions are required on multiple attributes, canceling the command will halt as soon as possible within the targeted command list.

The DmiMgmtCommand fields relevant to this command are **iCommand**, **iMgmtHandle**, and **iCmdHandle**. iCommand should be set to value **0x102**. iMgmtHandle should be set to the value originally assigned by the Service Layer when the application registered; it identifies the management application. iCmdHandle should be set to the value that identifies the handle of the outstanding command. The management application must ensure that the value of iCmdHandle is unique among all outstanding commands from that management application.

5.3. Registration

Before initiating any management activity through the DMI, management applications must register with the Service Layer. The registration process enables the Service Layer to provide services for the management application.

DmiRegisterMgmtReq

The command to register a management application uses the following request block:

Offset	*Size*	*Type*	*Variable Name*
0	64	STRUCT	DmiMgmtCommand
64	4	PTR	pConfirmFunc
68	4	PTR	pIndicationFunc

DmiMgmtCommand the command block. iCommand = **0x100**.

pConfirmFunc() pointer to a routine in the management application that the Service Layer will call upon completion of each command. The C prototype for this call is:

```
void pConfirmFunc(PTR cmd)
```

where cmd points to the Command block that has completed. If pConfirmFunc() is set to zero, the management application will not be able to use DmiInvoke (it cannot issue management directives), but may still receive indications if pIndicationFunc() is non-zero.

pIndicationFunc() pointer to a routine in the management application that the Service Layer will call to send unsolicited reports, either because of an event or the occurrence of a component (de)install. If the management application does not wish to get indications, it must set this field to zero. The C prototype for this call is:

```
void pIndicationFunc(DmiIndicate *buf)
```

where buf points to a DmiIndicate block.

DmiRegisterCnf

A confirm message will be sent if pConfirmFunc() is non-zero. The confirm message will contain the DmiRegisterCnf block as pointed to by pCnfBuf in the DmiMgmtCommand block. The confirm is of the form:

Offset	*Size*	*Type*	*Variable Name*
0	8	STRUCT	DmiVersion
8	4	INT	iDmiHandle

DmiVersion a block defining the version of the Service Layer.

iDmiHandle will be set to a unique handle for this management application on this system. This should be placed in the *iMgmtHandle* on all subsequent commands from this application.

Because the Service Layer keeps a management application's entry points until the application has unregistered, the application cannot provide different entry points for different commands. If this is a problem, there are two options. The first (and obvious) method is for the management application to register a dispatch routine as its entry point, and manually dispatch reports to the appropriate code. The second is to register the management application more than once, and give different entry points for each registration. Each registration will result in the Service

Layer assigning a new management handle, so the appropriate handle can be used when issuing a command. Each registration adds some housekeeping and processing overhead to the Service Layer, so this approach should be used only when the first method is not practical.

In addition to the management application's entry points, the Service Layer must keep the value of the osLanguage string in the command block. This value tells the Service Layer what language and character set to use when sending indications to the management application.

5.3.1. Unregistering

To unregister a management application from the Service Layer, send a DmiMgmtCommand block with iCommand = **0x101** with the appropriate handle. Applications that have registered more than once must unregister each assigned management command handle.

5.4. Indications

An *indication* is an unsolicited report — not that it is necessarily unwanted, only that the receiver was not expecting it. Registration provides a mechanism for management applications to register for notification of indications.

The Service Layer maintains a list of entry points for registered management applications. Each entry point is called when the Service Layer generates an indication. The called routine must immediately return control to the Service Layer, then perform any operations necessary based on the indication. Finally, the routine tells the Service Layer that it has finished processing the report by calling the function pResponseFunc() as given in the DmiIndicate block.

Each registered indication entry point is called for every indication. Thus each has to ignore the indications in which it is not interested.

Management applications should unregister their entry points before terminating; otherwise there may be unpredictable results, depending on the Service Layer implementation.

Indications are discussed further in the next chapter.

5.5. List commands

The List commands provide read access to a system's MIF database, to allow a management application to interrogate the manageable entities within a system. These commands allow a system to be probed to determine its characteristics without incurring the overhead of executing the code required to retrieve real-time values. All list commands are of the form *List*, *List First*, and *List Next*, which are used, respectively, for accessing lists randomly, from the start, or in sequential order. Specific List commands return information on individual entities. List First and List Next commands return data in a bulk fashion. In addition, commands are provided to retrieve the description fields from a system.

5.5.1. Listing components

List Component commands request a list of the components in a system. This command can be used to return all components in a system or only those with a certain group class. Calls for all components are referred to as 'unfiltered' and those for a specific group class as 'filtered'. There are three List Component commands:

DmiListComponentCmd (command code **0x200**) requests data for a specific component.

DmiListFirstComponentCmd (command code **0x201**) requests data for the first component in a system. "First" is defined as the component with the lowest component ID in that system. This will always be the Service Layer itself, which always has component Id of 1 (one). The Service Layer will fill in as many component definitions as the size of the confirm buffer allows.

DmiListNextComponentCmd (command code **0x202**) requests data for the component instrumentation whose ID is the next highest value beyond the one passed in the DmiListComponentCmd block. The Service Layer will fill in as many component definitions as the size of the confirm buffer allows.

DmiListComponentReq

All three List Component commands use the DmiListComponentReq block:

Offset	*Size*	*Type*	*Variable Name*
0	64	STRUCT	DmiMgmtCommand
64	4	INT	iComponentId
68	4	OFFSET	osClassString
72	4	INT	iGroupKeyCount
76	4	OFFSET	oGroupKeyList

DmiMgmtCommand the command block. The values of iCommand are

DmiListComponentCmd	**0x200**
DmiListFirstComponentCmd	**0x201**
DmiListNextComponentCmd	**0x202**

iComponentId component ID for the desired component. This field is ignored for DmiListFirstComponentCmd. For DmiListNextComponentCmd, the Service Layer will start filling the confirm buffer starting from the component ID that follows the value in this field. To start at the first component, set this field to zero. On return from the command, the Service Layer will update this field with the ID of the last component it put in the confirm buffer. The management application can continue issuing DmiListNextComponentCmd commands until the returned iStatus field indicates that there is no more data.

osClassString offset to a string that acts as a filter to retrieve data for components that have groups with specific class strings. The contents of this field look like class strings as previously described in the MIF chapter:

"***defining body*|*specific name*|*version***"

If a field of the class string is not important, it can be omitted. For example, if an application is looking for all component ID groups, it would use the string "DMTF|ComponentID|". Note that the version string is missing, so the Service Layer would consider any version a match, as long as the first two fields matched. If no filtering is desired, this field can be set to zero or to an empty class string ("||").

iGroupKeyCount count of the number of elements of type DmiGroupKeyData (described in the common data blocks section) pointed to by oGroupKeyList. If any keys are given, they are used with the group class for filtering. Thus the Service Layer will return a list of the components that have a particular row in a particular table. It is an error to specify keys without specifying a class string.

oGroupKeyList offset to a list of blocks of type DmiGroupKeyData.

The Service Layer does not contact any component instrumentation when performing List commands — it relies on the MIF database for all information. Thus when specifying keys that require components to be run before the values can be determined, the Service Layer will return the key as a *possible* match. Any matches from data satisfied from the MIF database will be

listed as *hard* matches. It is the responsibility of the management application to follow up with each component instrumentation if it wants to turn possible matches into hard matches.

DmiListComponentCnf

On return from this call, the confirm buffer contains an array of one or more DmiListComponentCnf blocks. iCnfCount in the DmiMgmtCommand block is set to the number of DmiListComponentCnf blocks returned from this call. iStatus indicates whether there was more data than would fit in the confirm buffer. In this case, the Service Layer updated iComponentId in the command buffer, so the management application can re-issue the command to continue getting data from where it left off.

The format of DmiListComponentCnf is:

Offset	*Size*	*Type*	*Variable Name*
0	4	INT	iComponentId
4	4	INT	iMatchType
8	4	OFFSET	osComponentName
12	4	INT	iClassListCount
16	4	OFFSET	oClassNameList
20	4	INT	iFileCount
24	4	STRUCT	DmiFileData[]

iComponentId handle assigned by the Service Layer when the component was installed.

iMatchType zero (0) if this is a definite match, and one (1) if the Service Layer could not determine because the key values required access to data controlled by component instrumentation (the List commands never execute component instrumentation). If the value is one (1), the management application will need to ask the component instrumentation.

osComponentName offset to a string that names the component.

iClassListCount count of the DmiClassNameData blocks pointed to by oClassNameList. The list represents the names of each class within this component's MIF.

oClassNameList offset to an array of DmiClassNameData blocks, defined as:

DmiClassNameData

Offset	*Size*	*Type*	*Variable Name*
0	4	INT	iGroupId
4	4	OFFSET	osClassString

iGroupId the ID of the group that corresponds to osClassString.

osClassString offset to a string that is the group's class string.

iFileCount count of the number of files in the MIF database for the component.

DmiFileData list of structures with the following format:

DmiFileData

Offset	*Size*	*Type*	*Variable Name*
0	4	INT	iFileType
4	4	OFFSET	oFileData

iFileType the code for the type of data that follows. The codes are documented in the component installation section below.

oFileData offset to an area of memory that contains the mapping data. The structure of this data (file name, memory buffer, etc.) depends on the type in iFileType.

5.5.2. Listing groups

List Group commands request a list of the groups in a component.

There are three forms of the List Group command:

DmiListGroupCmd (command code **0x204**) requests data for a specific group.

DmiListFirstGroupCmd (command code **0x205**) requests data for the first group in a specified component. *First* is defined as the group with the lowest group ID value in the specified component. The Service Layer will fill in as many groups as the size of the confirm buffer allows.

DmiListNextGroupCmd (command code **0x206**) requests data for the group whose ID is the next highest value beyond the one passed in the DmiListGroupReq block. The Service Layer will fill in as many groups as the size of the confirm buffer allows.

DmiListGroupReq

All three List Group commands use the DmiListGroupReq block:

Offset	*Size*	*Type*	*Variable Name*
0	64	STRUCT	DmiMgmtCommand
64	4	INT	iComponentId
68	4	INT	iGroupId

DmiMgmtCommand the command block. The values of iCommand are

DmiListGroupCmd **0x204**
DmiListFirstGroupCmd **0x205**
DmiListNextGroupCmd **0x206**

iComponentId ID of the desired component.

iGroupId ID of the desired group within the component identified by iComponentId. In DmiListFirstGroupCmd, this field is ignored. For the DmiListNextGroupCmd command, the Service Layer will start filling the confirm buffer starting from the group ID that follows the value in this field. To start at the first group, set this field to zero. On return from the command, the Service Layer will update this field with the ID of the last group it put in the confirm buffer. The management application can continue issuing DmiListNextGroupCmd commands until the returned iStatus field indicates that there is no more data.

DmiListGroupCnf

On return from this call, the confirm buffer contains an array of one or more DmiListGroupCnf blocks. iCnfCount in the DmiMgmtCommand block is set to the number of DmiListGroupCnf blocks returned from this call. iStatus indicates whether there was too much data than would fit in the confirm buffer. In this case, the Service Layer updated iGroupId in the command buffer so the management application can re-issue the command to continue getting data from where it left off.

The format of the DmiListGroupCnf block is:

Offset	*Size*	*Type*	*Variable Name*
0	4	INT	iGroupId
4	4	OFFSET	osGroupName
8	4	OFFSET	osClassString
12	4	INT	iGroupKeyCount
16	4	OFFSET	oGroupKeyList

iGroupId ID of the group.

osGroupName offset to a string that names the group.

osClassString offset to a string that names the class.

iGroupKeyCount count of the number of elements of type DmiGroupKeyData (described in the common data blocks section) pointed to by oGroupKeyList.

oGroupKeyList offset to a list of blocks of type DmiGroupKeyData. Note that within the DmiGroupKeyData blocks, no key *values* are returned (oKeyValue is set to zero) because the purpose of the List command is to identify the attributes that comprise the key.

5.5.3. Listing attributes

List Attribute commands request the attribute definitions within a group for a given component. They do not get the current attribute values.

There are three forms of the List Attribute command:

DmiListAttributeCmd (command code **0x208**) requests data for a specific attribute.

DmiListFirstAttributeCmd (command code **0x209**) requests data for the first attribute in a group. First is defined as the attribute with the lowest attribute ID in that group. The Service Layer will fill in as many attribute definitions as the size of the confirm buffer allows.

DmiListNextAttributeCmd (command code **0x20a**) requests data for the attribute whose ID is the next highest value beyond the one passed in the DmiListAttributeReq block. The Service Layer will fill in as many attribute definitions as the size of the confirm buffer allows.

DmiListAttributeReq

All List Attribute commands use the DmiListAttributeReq block:

Offset	*Size*	*Type*	*Variable Name*
0	64	STRUCT	DmiMgmtCommand
64	4	INT	iComponentId
68	4	INT	iGroupId
72	4	INT	iAttributeId

DmiMgmtCommand the command block. The values of iCommand are

DmiListAttributeCmd	**0x208**
DmiListFirstAttributeCmd	**0x209**
DmiListNextAttributeCmd	**0x20a**

iComponentId the ID of the desired component.

iGroupId the ID of the desired group within the component identified by iComponentId.

iAttributeId the ID of the desired attribute within the group. In DmiListFirstAttributeCmd, this field is ignored. For the DmiListNextAttributeCmd command, the Service Layer will start filling the confirm buffer starting from the attribute ID that follows the value in this field. To start at the first attribute, set this field to zero. On return from the command, the Service Layer will update this field with the ID of the last attribute it put in the confirm buffer. The management application can continue issuing DmiListNextAttributeCmd commands until the returned iStatus field indicates that there is no more data.

DmiListAttributeCnf

On return from this call, the confirm buffer contains an array of one or more DmiListAttributeCnf blocks. iCnfCount in the DmiMgmtCommand block is set to the number of DmiListAttributeCnf blocks returned from this call. iStatus indicates whether there was too much data than would fit in the confirm buffer. In this case, the Service Layer updated iAttributeId in the command buffer so the management application can re-issue the command to continue getting data from where it left off.

The format of the DmiListAttributeCnf block is:

Offset	*Size*	*Type*	*Variable Name*
0	4	INT	iAttributeId
4	4	OFFSET	osAttributeName
8	4	INT	iAttributeAccess
12	4	INT	iAttributeType
16	4	INT	iAttributeMaxSize
24	4	INT	iEnumListCount
28	4	OFFSET	oEnumList[]

iAttributeId ID of the attribute.

osAttributeName offset to a string that names the attribute.

iAttributeAccess access information for this attribute. Two pieces of information are encoded here. The three least significant bits indicate the read/write status. They can take the values:

0 unknown (usually indicates an error in the MIF database)
1 read-only
2 read-write
3 write-only
4 unsupported

The second piece of information is the database storage hint, encoded in the most significant bit. Possible values are:

0 specific (read-write information + 0)
1 common (read-write information + 0x80000000)

iAttributeType data type (Counter, Gauge, etc.). Legal values are:

0 unknown (usually indicates a MIF database error)
1 counter,
2 counter64
3 gauge
5 integer
6 integer64
7 octet string
8 displaystring, string
11 date

iAttributeMaxSize maximum size of the attribute. If unknown, this value returns -1. The size is in the units of iAttributeType for attributes.

iEnumListCount number of entries in the enumerations list.

oEnumList offset to a list of DmiEnumData blocks:

Offset	*Size*	*Type*	*Variable Name*
0	4	INT	iEnumValue
4	4	OFFSET	osEnumName

iEnumValue the integer value of the enumeration.

osEnumName offset to a literal string corresponding to the enumeration value.

5.5.4. Listing descriptions

These commands request the description field for the component, group or attribute specified. These calls are separated from the other list calls because the description fields may be of substantial size and may not be used or needed in some cases.

There are three commands in this group: ***DmiListComponentDescCmd***, ***DmiListGroupDescCmd***, and ***DmiListAttributeDescCmd***, for requesting the descriptions of components, groups, and attributes, respectively.

DmiListDescReq

The three List Description commands use the DmiListDescReq block:

Offset	*Size*	*Type*	*Variable Name*
0	64	STRUCT	DmiMgmtCommand
64	4	INT	iComponentId
68	4	INT	iGroupId
72	4	INT	iAttributeId
76	4	INT	iOffset

DmiMgmtCommand the command block. The values of iCommand are

DmiListComponentDescCmd	**0x203**
DmiListGroupDescCmd	**0x207**
DmiListAttributeDescCmd	**0x20b**

iComponentId the ID of the desired component.

iGroupId the ID of the desired group within the component identified by iComponentId. It is ignored when listing component descriptions.

iAttributeId the ID of the desired attribute within the group. It is only required for getting attribute descriptions, and ignored otherwise.

iOffset number of characters into the description where the Service Layer should start fetching the description string. Use zero to start at the beginning of the description. On return, the Service Layer will update this field with the index of the last character written into the confirm buffer, and set iStatus accordingly. If iStatus indicates that there are more characters to come, the application can re-issue the call to get more of the description text.

On return from this call, the confirm buffer contains the text from the description associated with the entity requested. iCnfCount in the command block is always set to 1.

5.6. Getting attributes

The difference between **List** and **Get** is that a List command returns MIF information about an object, while a Get command requests the actual value for an object.

DmiGetAttributeReq

The DmiGetAttributeCmd command (command code **0x300**) requests the current values of attributes within groups. It uses the DmiGetAttributeReq block:

Offset	*Size*	*Type*	*Variable Name*
0	64	STRUCT	DmiMgmtCommand
64	4	INT	iComponentId
68	16	STRUCT	DmiGetAttributeData[]

DmiMgmtCommand the command block. iCommand = **0x300**.

iComponentId the ID of the desired component.

DmiGetAttributeData one or more DmiGetAttributeData blocks. The caller must indicate the number of appended blocks in the iRequestCount element in the DmiMgmtCommand block. The DmiGetAttributeData block is:

DmiGetAttributeData

Offset	*Size*	*Type*	*Variable Name*
0	4	INT	iGroupId
4	4	INT	iGroupKeyCount
8	4	OFFSET	oGroupKeyList
12	4	INT	iAttributeId

iGroupId the ID of the desired group.

iGroupKeyCount count of the number of DmiGroupKeyData blocks pointed to by oGroupKeyList.

oGroupKeyList offset to a list of DmiGroupKeyData blocks. iGroupKeyCount is the number of blocks in the list.

iAttributeId the ID of the desired attribute.

DmiGetAttributeCnf

On return from this call, the confirm buffer contains an array of one or more DmiGetAttributeCnf blocks. iCnfCount in the command block lists the number of DmiGetAttributeCnf blocks.

Offset	*Size*	*Type*	*Variable Name*
0	4	INT	iGroupId
4	4	INT	iAttributeId
8	4	INT	iAttributeType
12	4	OFFSET	oAttributeValue

iGroupId the ID of the group to which the attribute belongs.

iAttributeId the ID of the attribute.

iAttributeType data type (Counter, Gauge, etc.). Legal values are:

0	unknown (usually indicates a MIF database error)
1	counter
2	counter64
3	gauge
5	integer
6	integer64
7	octet string
8	displaystring, string
11	date

oAttributeValue offset to the value returned from this get operation. If no value is provided (for instance, in the case of write-only attributes), the offset will be zero.

5.7. Setting attributes

To enable concurrent access by multiple management applications, the DMI allows more than one pass through a list of set commands. The first is a call to *reserve* the attributes, that tests a component to see if a set operation on a given attribute could occur. This reserve command asks the component instrumentation to validate a set without performing the set operation. The component instrumentation must validate the parameters of the call and reserve any resources that would be required if the set command were issued. A positive confirm from this call says the component instrumentation has validated the parameters and reserved any resources that may be required to perform the set. A negative confirm from this call says the component instrumentation is unable to perform the set

If all the attributes in the list returned successfully from the reserve operation, the management application would typically issue a *set* for each attribute. If a reserve failed, the management application may want to *release* each of the attributes it previously reserved. Releasing allows the component instrumentation to de-allocate any resources it had acquired. The task of *roll-back* (setting attributes back to their old values) is the responsibility of the management application.

DmiSetAttributeReq

There are three Set calls; ***Set***, ***Reserve*** and ***Release***. Each uses the DmiSetAttributeReq block:

Offset	*Size*	*Type*	*Variable Name*
0	64	STRUCT	DmiMgmtCommand
64	4	INT	iComponentId
68	20	STRUCT	DmiSetAttributeData[]

DmiMgmtCommand the command block. The values of iCommand are

DmiSetAttributeCmd	**0x301**
DmiSetReserveAttributeCmd	**0x302**

DmiSetReleaseAttributeCmd **0x303**

iComponentId the ID of the component to be operated on.

DmiSetAttributeData one or more DmiSetAttributeData blocks. The caller must indicate the number of appended blocks in the iRequestCount element in the DmiMgmtCommand block. The DmiSetAttributeData block looks like:

SmiSetAttributeData

Offset	*Size*	*Type*	*Variable Name*
0	4	INT	iGroupId
4	4	INT	iGroupKeyCount
8	4	OFFSET	oGroupKeyList
12	4	INT	iAttributeId
16	4	OFFSET	oAttributeValue

iGroupId the ID of the desired group.

iGroupKeyCount count of the number of DmiGroupKeyData blocks pointed to by oGroupKeyList.

oGroupKeyList offset to a list of DmiGroupKeyData blocks. iGroupKeyCount is the number of blocks in the list.

iAttributeId the ID of the desired attribute.

oAttributeValue offset to the value to be set.

There is no data returned from this call in the confirm buffer. The value of iStatus indicates either success or failure. In the case of a failure condition, the value of (iCnfCount + 1) corresponds to the number of the command block that caused the error.

5.7.1. Reserving attributes

Reserving an attribute will test to see that the set is a valid operation. It ensures that a later set would be successfully issued. Note that it is NOT a locking operation; management applications can never assume exclusive control of a component.

This call uses the DmiSetAttributeReq block, with the command code **0x302**.

The component instrumentation receiving the Reserve command is responsible for testing the arguments to see that the set can be done. Testing involves ensuring that the arguments are within range, and allocating and reserving any resources that may be needed to ensure the success of a later set.

After a successful set, the component instrumentation will automatically release any resources that it had reserved. The management application does not need to explicitly issue a Release command after a set.

The component instrumentation requires that all Reserve commands will be followed by either Set or Release commands to perform the set or release the reserved resources. The failure of the management application to issue either command is considered a catastrophic failure and is outside the scope of the component instrumentation's responsibility to try to correct.

If a management application requests that multiple attributes be reserved and one of the reservations fails, the Service Layer will return the attribute where the command failed. It is up to the management application to release the successfully reserved attributes.

5.7.2. Releasing attributes

The Release command is used when the caller wishes to decommit from a set operation after a reserve has been issued. The Release command uses the DmiSetAttributeReq block with the command code **0x303**.

5.8. Getting rows

Get Row commands request the values of all the attributes in a particular group. Because groups can be a scalar set of attributes or can be arranged as tables -- arrays of sets of attributes -- it is more convenient to think of them as *rows*. Groups used as tables include keys to specify the desired instance (or row) of the group.

There are three forms of the Get Row command:

DmiGetRowCmd (command code **0x304**) requests the values of attributes from a specific group, or specific row of a table.

DmiGetFirstRowCmd (command code **0x305**) requests the values of the attributes in the first row of a table. This is the same as DmiGetRowCmd for non-tabular groups.

DmiGetNextRowCmd (command code **0x306**) requests the values of the attributes of the row after the row whose key is passed in the command block.

DmiGetRowReq

All three Get Row commands use the DmiGetRowReq block:

Offset	*Size*	*Type*	*Variable Name*
0	64	STRUCT	DmiMgmtCommand
64	4	INT	iComponentId
68	4	INT	iGroupId
72	4	INT	iGroupKeyCount
76	4	OFFSET	oGroupKeyList
80	4	INT	iAttributeId

DmiMgmtCommand the command block. The values of iCommand are

DmiGetRowCmd	**0x304**
DmiGetFirstRowCmd	**0x305**
DmiGetNextRowCmd	**0x306**

iComponentId the ID of the desired component.

iGroupId the ID of the desired group.

iGroupKeyCount count of the number of DmiGroupKeyData blocks pointed to by oGroupKeyList.

oGroupKeyList offset to a list of DmiGroupKeyData blocks. iGroupKeyCount is the number of blocks in the list.

iAttributeId the attribute ID just before the attribute ID where the Get Row command should start. Use zero to start at the first attribute in the group. This is usually used when the confirm buffer is too small and the Service Layer has more information than will fit in the buffer. Upon return the Service Layer updates this field with the last attribute returned, and sets iStatus to one (1) to indicate that there is more information than would fit in the confirm buffer.

DmiGetRowCnf

On return from this call, the confirm buffer contains a DmiGetRowCnf block followed by one or more DmiGetAttributeCnf blocks. The format of DmiGetRowCnf is:

Offset	*Size*	*Type*	*Variable Name*
0	4	INT	iGroupId
4	4	INT	iGroupKeyCount
8	4	OFFSET	oGroupKeyList
12	4	INT	iAttributeCount
16	12	STRUCT	DmiGetAttributeCnf[ ]

iGroupId the ID of the group.

iGroupKeyCount count of the number of DmiGroupKeyData blocks pointed to by oGroupKeyList.

oGroupKeyList offset to a list of DmiGroupKeyData blocks. iGroupKeyCount is the number of blocks in the list.

iAttributeCount count of the number of DmiGetAttributeCnf blocks immediately following this block.

DmiGetAttributeCnf the list of DmiGetAttributeCnf blocks.

In the case where the size of a group exceeds the size of the caller's confirm buffer, the caller will get back a status of one (1) indicating that the command was successful but that there is more of the group to be retrieved. In this case, the command can be re-issued to retrieve the rest of the group. To do this, the caller sets the iAttributeId field to the value of the last attribute returned in the previous Get Row command. The component instrumentation will return the attribute values that follow this attribute.

Note for component providers

Note that the command can be asking for key values or for attribute values. To determine which, the component instrumentation must compare the iGroupKeyCount field of DmiGetRowReq to the iGroupKeyCount field of DmiGetRowCnf. If the key count in the confirm block is less that the key count of the request block, the Service Layer is requesting that the next key value be returned.

The key value to return for the Get First Row command is the value of the next key attribute in the first row of the table. For the Get Next Row command, the key values in the DmiGetRowReq structure should be inspected to determine the current row; then the value in the succeeding row should be returned.

The DmiGetRowCnf structure is *always* built by the Service Layer, and oGroupKeyList specifies the start of DmiGroupKeyData structures. The pCnfBuf member of DmiCiCommand points to the position in the confirm buffer where the key value should be placed. The iGroupKeyCount should be incremented, but the iCnfCount member of the DmiMgmtCommand should NOT be.

5.9. Implementation-specific commands

Service Layers may optionally accept non-standard command codes; these commands are defined by the SL implementation and exist in the range above **0x80000000** (the high bit of the command is set). Implementation-specific commands are used (invoked, canceled) like any standard command.

Implementation-specific commands are available if the SL provider wants to provide additional functionality in the Service Layer. It is up to the SL provider to define and document the commands, command blocks, confirm blocks and status codes.

Because these non-standard commands can differ among SL implementations, a management application should not use these commands until it verifies the revision of the Service Layer in the DmiVersion block returned on registration.

6. The Component Interface

Calls from the Service Layer through the CI use only attribute, group and key identifiers for naming. By the time the component instrumentation is reached, the Service Layer has resolved the component ID. Calls from the component instrumentation through the CI (indications) additionally need the component ID so the SL can identify the component instrumentation.

The mechanics of the communication from the Service Layer to the component instrumentation are specific to the operating system environment. The two functions (entry points) that the component instrumentation must respond to are **DmiCiInvoke()** and **DmiCiCancel()**. These are described below.

The Service Layer is responsible for launching the component instrumentation if necessary and passing along the command blocks. The launching mechanism is implementation-specific.

6.1. Installation

A component makes itself known to the management system by "installing" a MIF file into the MIF database. It needs to install the MIF file only once - when the component is first introduced to the system.

A MIF is installed and removed using the DmiCiInstall and DmiCiUninstall commands. These are the only primitives provided to manipulate the (implementation-specific) MIF database; there are no MIF database modification mechanisms. If an installed MIF requires modification, it must be removed, modified, and then re-installed.

Upon installation or removal of a MIF, the Service Layer will issue an indication to all registered management applications. See the indication/event section that follows.

The installation can optionally include files that provide language mapping and mapping logic between the MIF and other management information standards.

DmiCiInstallData

To install a new MIF file, the component instrumentation fills out a DmiCiInstallData block and sends it to the Service Layer with DmiInvoke(). The command block looks like:

Offset	*Size*	*Type*	*Variable Name*
0	64	STRUCT	DmiMgmtCommand
64	4	INT	iComponentId
68	4	INT	iFileCount
72	8	STRUCT	DmiFileData[ ]

DmiMgmtCommand the command block. iCommand = **0x402**. iMgmtHandle should always be set to zero.

iComponentId the Service-Layer defined ID of this component. On initial install, this should be set to zero. On subsequent installs - used to provide additional MIF data for this component, such as MIF files for additional languages - this should be set to the component's ID.

iFileCount number of DmiFileData structures that follow.

DmiFileData a list of structures with the following format:

DmiFileData

Offset	*Size*	*Type*	*Variable Name*
0	4	INT	iFileType
4	4	OFFSET	osFileData

iFileType type for this component's MIF files. The type codes are 32-bit values with the following semantics: if the value is even, the "file" that follows is a file name. If the value is odd, the "file" is a pointer to the MIF data in memory.

The following type codes are defined:

Type Code	*Meaning*
0	reserved, do not use
1	reserved, do not use
2	MIF file name
3	MIF file pointer
4	SNMP mapping file name
5	SNMP mapping file pointer
0x80000000	start of implementation-specific types

Note that like command codes, file type codes above **x80000000** are reserved for implementation-specific use.

Also note that as of this specification, the format for the SNMP mapping file has not been standardized.

osFileData offset to a string whose contents depend on the type code above.

On successful return, **iComponentId** is set to a numeric ID assigned by the Service Layer. This ID uniquely identifies the component described by the installation. This system-specific ID is stored by the Service Layer in the MIF database. The MIF database is a persistent data store, so the component ID, once assigned, remains the same until the component is removed. The component should remember this component ID, as it is used when issuing indications or when the component is removed from the system.

Upon successful installation, the Service Layer sends an indication to every management application that is currently registered with it. If the installation created a new component, the value of iIndicationType (in the DmiIndicate block) is **2.** If the installation resulted in any additional MIFs being added to an existing component, the value of iIndicationType is **4.** In both cases, the indication data is a DmiListComponentCnf block. If the installation was

6.2. Removal

Removing a component from the MIF database will effectively remove the component from further participation in the management system. In the DMI, removing a component is called "uninstalling."

Removing a component from the MIF database does not mean the component is not present in the computer system; only that the management system does not know about it.

DmiCiUninstallData

To remove a component from the MIF database, the component instrumentation fills out a DmiCiUninstallData block and sends it to the Service Layer with DmiInvoke(). The command block looks like:

Offset	*Size*	*Type*	*Variable Name*
0	64	STRUCT	DmiMgmtCommand
64	4	INT	iComponentId

DmiMgmtCommand the command block. iCommand = **0x403**. iMgmtHandle should always be set to zero.

iComponentId the Service-Layer defined ID of this component.

Upon successful removal, the Service Layer sends an indication to every management application that is currently registered with it. The value of iIndicationType (in the DmiIndicate block) is **3**, and the indication data is a DmiListComponentCnf block.

6.3. Invoking a command

The component instrumentation must provide an entry point so the Service Layer can send it command blocks for processing. The SL will call the component instrumentation entry point ***DmiCiInvoke***(). The C-language prototype for this call is:

unsigned long DmiCiInvoke(PTR command)

where command is the complete command block.

The return result from the component instrumentation is a 32-bit status value indicating success or failure. The possible status values are described in Chapter 8.

6.4. Canceling

The Service Layer may notify the component instrumentation of a desire to cancel (terminate processing on) an individual command by calling the ***DmiCiCancel()*** entry point within the component instrumentation. This may be issued as a result of a management application issuing a DmiCancelCmd command to the Service Layer. The call is:

unsigned long DmiCiCancel(void)

No parameters are needed since by definition the component instrumentation can only be processing one command at a time.

The return result from the component instrumentation should be zero for success and non-zero for any error. The Service Layer will place the result in iStatus when sending the confirm block to the management application.

6.5. Registration

Component instrumentation code may register with the Service Layer. Registration is desirable when the component instrumentation code is already running (such as management code associated with a running device driver). Registering management entry points within the running code is called registering the ***direct*** interface. The direct interface is also useful when the time required to locate and load an executable file, set up and tear down a process would be excessive.

Component instrumentation that has registered a direct interface causes the SL to override its current access mechanism for the registered attributes. Instead of manipulating the data in the MIF database or invoking programs, it will call the entry point provided in the registration call. Once the component unregisters, the SL 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.

DmiRegisterCiInd

The DmiRegisterCiInd block can be used:

- to register a callable interface for components that have resident instrumentation code
- to get the version of the Service Layer.

DmiRegisterCiCmd uses the DmiRegisterCiInd block:

Offset	*Size*	*Type*	*Variable Name*
0	64	STRUCT	DmiMgmtCommand
68	4	INT	iComponentId
72	4	PTR	pAccessFunc
76	4	PTR	pCancelFunc
80	4	INT	iAccessListCount
84	8	STRUCT	DmiAccessData[ ]

DmiMgmtCommand the command block. iCommand = **0x400**.

iComponentId identifier the Service Layer assigned to the component on installation.

pAccessFunc pointer to the entry point of the component instrumentation. The Service Layer will call the component instrumentation with the C prototype:

unsigned long pAccessFunc(PTR command)

where command is the complete block. A value of zero is illegal.

pCancelFunc pointer to the entry point the SL should call to cancel an outstanding operation though the direct interface. The Service Layer will call with the C prototype:

unsigned long pCancelFunc(void)

A value of zero is illegal.

iAccessListCount count of the number of blocks of type DmiAccessData, as defined below.

DmiAccessData the groups and/or individual attributes that use the direct interface. The format of the DmiAccessData block is:

DmiAccessData

Offset	*Size*	*Type*	*Variable Name*
0	4	INT	iGroupId
4	4	INT	iAttributeId

iGroupId group that uses the high performance interface. A value of zero indicates that all groups within this MIF use the high performance interface, and the following iAttributeId field is ignored.

iAttributeId attributes, within the group specified by *iGroupId*, that use the high performance interface. A value of zero indicates that all attributes within this group use the high performance interface.

DmiRegisterCnf

Upon return from this call, the Service Layer will place a DmiRegisterCnf block in the response buffer. This block is the same as described in the section on management registration:

Offset	*Size*	*Type*	*Variable Name*
0	8	STRUCT	DmiVersion
8	4	INT	iDmiHandle

DmiVersion a block defining the version of the Service Layer.

iDmiHandle will be set to a unique handle for this component instrumentation. This is only valid for as long as the component instrumentation is registered. This handle should be passed to the Service Layer when unregistering.

A component may register more than once. This is necessary when a component has multiple entry points for instrumentation. In this case it will end up with multiple handles.

DmiUnregisterCiInd

DmiUnregisterCiCmd (command code **0x401**) tells the Service Layer to remove a direct component instrumentation interface from the Service Layer's table of registered interfaces. The call uses the DmiUnregisterCiInd block:

Offset	*Size*	*Type*	*Variable Name*
0	64	STRUCT	DmiMgmtCommand
64	4	INT	iCiHandle

DmiMgmtCommand the command block. iCommand = **0x401**.

iCiHandle handle the Service Layer assigned to the component on registration.

Components must unregister once for each registration performed.

No return buffer is required or used.

6.6. Sending events

Component instrumentation may send unsolicited messages to the Service Layer to notify it of some particular situation. These unsolicited messages are known as *events*. When they reach a management application they are usually known as *indications*, with indication id **1**. Events are often used to describe a catastrophic occurrence or other activity that should be made known quickly. The event identification is specific to a given component. Events look like unsolicited *gets*, so event structures are specified as groups in the component's MIF file.

Because the DMI is a data interface, all commands are specified with data blocks. However, there is one function call provided so the component instrumentation can send the indication block to the Service Layer. The ***DmiIndicate***() call sends the indication (as defined by the DmiIndicate block and any following blocks) to the Service Layer for processing. The C-language prototype for this call is:

```
unsigned long  DmiIndicate(PTR command)
```

where command is the complete command block. The return result is a 32-bit status value indicating success or failure. The possible status values are described in Chapter 8.

The Service Layer will immediately return control to the component instrumentation while it processes the indication. The component instrumentation is free issue additional indications, but it cannot re-use the indication buffer it handed to the Service Layer -- simultaneous indications from a component instrumentation must use different indication blocks. When the Service Layer is done with the indication, it calls the function pResponseFunc() that was given in the original indication block. At that point, the component instrumentation can re-use the block.

Note that in some operating environments, it is theoretically possible to get a response message before DmiIndicate() returns, due to the asynchronous nature of the DMI. Service Layer implementations in such environments should document the possible behavior.

DmiEventData

To send an event, the component instrumentation sends a DmiIndicate block to the Service Layer's **DmiIndicate()** entry point. The event data is defined in a DmiEventData block that maps onto the oIndicationData field in the DmiIndicate block. It looks like:

Offset	*Size*	*Type*	*Variable Name*
0	4	INT	iClassCount
4	4	STRUCT	DmiClassData[ ]

iClassCount the number of group class strings in the component.

DmiClassData a list of the group class data blocks for this component:

DmiClassData

Offset	*Size*	*Type*	*Variable Name*
0	4	INT	iComponentId
4	4	OFFSET	osClassString
8	4	INT	iRowCount
12	4	STRUCT	DmiGetRowCnf[ ]

iComponentId the ID of the component that this event is about.

osClassString offset to the string that names the group class for the event.

iRowCount number of DmiRowList structures below.

DmiGetRowCnf a list of one or more DmiGetRowCnf structures that describes the data in the event.

Component instrumentation code is encouraged to pass numeric values for status indications rather than textual strings. If the status codes are passed as enumeration values, the Service Layer can translate them to the appropriate language mappings before sending them to the registered management applications. If component instrumentations send text strings, no language mapping will be possible -- possibly confusing the (human) operators.

No return buffer is required or used.

7. Compliance

A Service Layer may consider itself "DMI compliant" if it implements this specification in its entirety except for the optional security and internationalization features.

The Service Layer must accept (but not necessarily use) the **language** statement in the MIF and the osLanguage element in the DmiMgmtCommand block. If the Service Layer does not support Unicode, it must issue a warning when encountering a Unicode MIF.

8. Command and Status Codes

8.1. Command codes

In the following table, the notation "*" means that the block may appear more than once. Because most commands use or embed the DmiMgmtCommand and DmiCiCommand request blocks and (on return) the DmiConfirm blocks, they have been omitted from the table.

MI Registration & Cancel Commands:

Code	*Command*	*Request Blocks Used*	*Confirm Blocks Used*
0x100	DmiRegisterMgmtCmd	DmiRegisterMgmtReq	DmiRegisterCnf DmiVersion
0x101	DmiUnregisterMgmtCmd		
0x102	DmiCancelCmd		

MI List Commands:

Code	*Command*	*Request Blocks Used*	*Confirm Blocks Used*
0x200	DmiListComponentCmd	DmiListComponentReq DmiGroupKeyData*	DmiListComponentCnf* DmiClassNameData* DmiFileData*
0x201	DmiListFirstComponentCmd	DmiListComponentReq DmiGroupKeyData*	DmiListComponentCnf* DmiClassNameData* DmiFileData*
0x202	DmiListNextComponentCmd	DmiListComponentReq DmiGroupKeyData*	DmiListComponentCnf* DmiClassNameData* DmiFileData*
0x203	DmiListComponentDescCmd	DmiListDescReq	
0x204	DmiListGroupCmd	DmiListGroupReq	DmiListGroupCnf* DmiGroupKeyData*
0x205	DmiListFirstGroupCmd	DmiListGroupReq	DmiListGroupCnf* DmiGroupKeyData*
0x206	DmiListNextGroupCmd	DmiListGroupReq	DmiListGroupCnf* DmiGroupKeyData*
0x207	DmiListGroupDescCmd	DmiListDescReq	
0x208	DmiListAttributeCmd	DmiListAttributeReq	DmiListAttributeCnf* DmiEnumData*
0x209	DmiListFirstAttributeCmd	DmiListAttributeReq	DmiListAttributeCnf* DmiEnumData*
0x20a	DmiListNextAttributeCmd	DmiListAttributeReq	DmiListAttributeCnf* DmiEnumData*
0x20b	DmiListAttributeDescCmd	DmiListDescReq	

MI Get and Set Commands:

Code	*Command*	*Request Blocks Used*	*Confirm Blocks Used*
0x300	DmiGetAttributeCmd	DmiGetAttributeReq DmiGetAttributeData* DmiGroupKeyData*	DmiGetAttributeCnf*
0x301	DmiSetAttributeCmd	DmiSetAttributeReq DmiSetAttributeData* DmiGroupKeyData*	
0x302	DmiSetReserveAttributeCmd	DmiSetAttributeReq DmiSetAttributeData* DmiGroupKeyData*	
0x303	DmiSetReleaseAttributeCmd	DmiSetAttributeReq DmiSetAttributeData* DmiGroupKeyData*	
0x304	DmiGetRowCmd	DmiGetRowReq DmiGroupKeyData*	DmiGetRowCnf DmiGroupKeyData* DmiGetAttributeCnf*
0x305	DmiGetFirstRowCmd	DmiGetRowReq DmiGroupKeyData*	DmiGetRowCnf DmiGroupKeyData* DmiGetAttributeCnf*
0x306	DmiGetNextRowCmd	DmiGetRowReq DmiGroupKeyData*	DmiGetRowCnf DmiGroupKeyData* DmiGetAttributeCnf*

CI Commands:

Code	*Command*	*Indication Blocks Used*	*Response Blocks Used*
0x400	DmiRegisterCiCmd	DmiRegisterCiInd DmiAccessData*	DmiRegisterCnf DmiVersion
0x401	DmiUnregisterCiCmd	DmiUnregisterCiInd	
0x402	DmiCiInstallCmd	DmiCiInstallData DmiFileData*	
0x403	DmiCiUninstallCmd	DmiCiUninstallData	
NA	DmiEvent	DmiIndicate DmiEventData DmiClassData* DmiGetRowCnf*	

Implementation-Specific Commands:

Code	*Command*	*Command Blocks Used*	*Confirm Blocks Used*
0x80000000	Start of SL-specific commands	Implementation dependent	Implementation dependent

8.2. Status 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 Layer 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.

Non-error condition codes:

Value	*Description*
0x00000	Success
0x00001	More data is available

Database errors:

Value	*Description*
0x00100	Attribute not found
0x00101	Value exceeds maximum size
0x00102	Component instrumentation not found
0x00103	Enumeration error
0x00104	Group not found
0x00105	Illegal keys
0x00106	Illegal to set
0x00107	Can't resolve attribute function name
0x00108	Illegal to get
0x00109	No description
0x0010a	Row not found
0x0010b	Direct interface not registered
0x0010c	MIF database is corrupt
0x0010d	Attribute is not supported

Service Layer errors:

Value	*Description*
0x00200	Buffer full
0x00201	Ill-formed command
0x00202	Illegal command
0x00203	Illegal handle
0x00204	Out of memory
0x00205	No confirm function
0x00206	No response buffer
0x00207	Command handle is already in use
0x00208	DMI version mismatch
0x00209	Unknown CI registry
0x0020a	Command has been canceled
0x0020b	Insufficient privileges
0x0020c	No access function provided
0x0020d	OS File I/O error
0x0020e	Could not spawn a new task
0x0020f	Ill-formed MIF
0x00210	Invalid file type
0x00211	Service Layer is inactive
0x00212	UNICODE not supported

Other errors[3]:

Value	*Description*
0x01000	Start of DOS-specific status codes
0x02000	Start of Windows-specific status codes
0x03000	Start of OS/2-specific status codes
0x04000	Start of UNIX-specific status codes
0x05000	Start of MacOS-specific status codes
0x06000	Start of Windows NT-specific status codes
0x10000	Start of component-specific status codes

[3]The operating system names given in this table are trademarks of their respective trademark holders.

9. MIF Grammar

The MIF grammar, given in BNF notation, is given below:

```
<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> }* ]
                                    (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

<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> ]
                                        { <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>

<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 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> ]
                               (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 |
Unsupported

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

<Storage Type> ::= Specific |
Common

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

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

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

<Value Statement> ::= Value '=' <Constant Expression>

<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 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>] <Unsigned Integer>
<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 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"

A. ISO 639

Technical contents of ISO 639:1988 (E/F) "Code for the representation of names of languages".

This is not the official ISO document. It is also not part of the DMI specification, but is here for reference.

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

B. ISO 3166

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.

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.

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 and Barbuda	AG	Costa Rica	CR		
Argentina	AR	Cote D'Ivoire	CI	Haiti	HT
Aruba	AW	Cuba	CU	Heard and Mc Donald 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 of)	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 Islands	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 and 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 Islands	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		

C. Sample MIF

This appendix is not part of the DMI specification. It is here only for reference.

```
//
//      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|1.0"       // 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 = 0
End Attribute

         End Group                              // DMTF|ComponentID|1.0
```

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

        Start Group
Name = "Service Group"
Id = 2
Class = "AnyComputer|SystemGroup|x001"

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 = "Warrantee 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 = "Warrantee 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,"
                          " like 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
```

D. Glossary

This appendix is not part of the DMI specification. It is here only for reference.

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.
CMOL	CMIP Over LLC, a lightweight version of CMIP, standardized by the IEEE.
Command Block	The concatenation of data blocks (data structures) that constitute a command to be sent between *management applications* and the *Service Layer* and between the Service Layer 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 Layer* puts response data.
Direct Interface	Method by which a *component instrumentation* informs the *Service Layer* that it (the component instrumentation) is already running. Rather than starting the code to service incoming requests, the Service Layer will use the already running code.
DMI	Desktop Management Interface, the subject of this specification.
Event	A type of *indication* (unsolicited report) that originates from a *component instrumentation.*
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 Layer*, or from the Service Layer to a *management application*.
ISO 8859-1	A character encoding standard defined by ISO. Commonly known as extended ASCII or 8-bit ASCII.
Key	An identifier of a particular instance (row) of a table.
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 Layer.

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 Layer* (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.
Request	A command with associated context issued from the *management application* to accomplish management.
Response	The final response from an *Indication*.
Row	An instance of a *table*.
Service Layer (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.
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.

E. Index

A

access statement....22, 23, 39
alignment 11
asynchronous 7, 9, 11, 32, 52
attribute definition 21, 22, 24, 38
attribute ordering 13
attributes....5, 14, 20, 23, 38, 44, 71

B

buffer management 11, 26

C

cancel 11, 27, 32, 49, 50
case sensitivity 14
class name 21, 35, 71
class statement....21, 24
command codes 54
command sequencing....11, 12, 13
comments 14
common keyword....23, 39
compliance 53
component definition 17, 19
component ID 35, 47, 48
component instrumentation..... 4, 7, 8, 9, 11, 23, 47, 71
Component Interface....4, 47, 71
ComponentID group 25, 35
confirm....7, 9, 32, 33
confirm buffer.... 10, 11, 44, 45, 71
counter.... 15, 30, 40, 42
counter64....15

D

date.... 15, 30, 40, 42
default values.... 20, 23, 24, 25
description statement 18, 20
direct interface....49, 51, 71
displaystring 15, 30, 40, 42
DmiAccessData 50, 55
DmiCancel....12
DmiCancelCmd 49, 54
DmiCiCancel....47, 49
DmiCiCommand.... 10, 26, 28, 45, 54
DmiCiInstall....47
DmiCiInstallCmd 55
DmiCiInstallData....47
DmiCiInvoke 47, 49
DmiCiUninstall 47
DmiCiUninstallCmd....55
DmiCiUninstallData 48
DmiClassNameData....36, 54
DmiConfirm 28, 54
DmiEnumData....40, 54
DmiEvent 55
DmiEventData 29, 52
DmiGetAttributeCmd....41, 55
DmiGetAttributeCnf 41, 45, 55
DmiGetAttributeData....28, 41, 55
DmiGetAttributeReq 41, 55
DmiGetFirstRowCmd 44, 55
DmiGetNextRowCmd 44, 55
DmiGetRowCmd....44, 55
DmiGetRowCnf 45, 55
DmiGetRowReq 44, 45, 55
DmiGroupKeyData 30, 35, 38, 41, 43, 44, 45, 54, 55
DmiIndicate 12, 29, 33, 34, 48, 49, 52, 55
DmiInvoke 12, 32, 33, 47
DmiListAttributeCmd 38, 54
DmiListAttributeCnf....39, 54
DmiListAttributeDescCmd....40, 54
DmiListAttributeReq....38, 54
DmiListComponentCnf.... 29, 36, 48, 49, 54
DmiListComponentDescCmd....40, 54
DmiListComponentReq....35, 54
DmiListDescReq 54
DmiListFirstAttributeCmd 38, 39, 54
DmiListFirstComponentCmd 34, 35, 54
DmiListFirstGroupCmd 37, 54
DmiListGroupCmd 37, 54
DmiListGroupCnf....37, 38, 54
DmiListGroupDescCmd....40, 54
DmiListGroupReq....10, 37, 54
DmiListNextAttributeCmd....38, 39, 54
DmiListNextComponentCmd....35, 54
DmiListNextGroupCmd....37, 54
DmiMgmtCommand 10, 26, 28, 32, 34, 54
DmiRegisterCiCmd.... 55
DmiRegisterCiInd....50, 55
DmiRegisterCnf....33, 50, 54, 55
DmiRegisterMgmtReq 54
DmiSetAttributeCmd 42, 55
DmiSetAttributeData 28, 43, 55
DmiSetAttributeReq....42, 43, 44, 55
DmiSetReleaseAttributeCmd....43, 44, 55
DmiSetReserveAttributeCmd 42, 43, 55
DmiTimeStamp 29, 30
DmiUnregisterCiCmd 55
DmiUnregisterCiInd 51

DmiVersion 30, 33, 51, 54, 55

E

entry points 12, 47
enumerations 17, 40
events 6, 29, 51, 71

F

floating point 16

G

gauge 15, 30, 40, 42
get commands 9, 44, 45, 55
group definition 20
groups 5, 20, 21, 24, 37, 51, 71

H

handle 27, 33, 34, 36, 51

I

id statement 19, 20, 22, 24
implementation dependencies ..26, 27, 39, 45, 47, 48, 55
indications 6, 8, 13, 29, 34, 51, 71
installation 47, 48
integer 14, 15, 30, 40, 42
integer64 15
ISO 3166 ..... 18, 64
ISO 639 ..... 18, 63
ISO 8859-1 ..... 10, 14, 16, 18, 30

K

key statement 21
keys 5, 30, 71

L

language 27, 47
language statement 17, 18, 53
list commands 9, 34, 37, 38, 54
literal escapes 16
literal strings 14, 16, 23
locking 11, 43

M

management application 4, 71
Management Interface 5, 32, 71
management protocol 4, 5, 19, 71, 72
MIF database 9, 13, 34, 47, 72
MIF files 4, 14, 17, 47, 48

N

name statement 18, 19, 24

O

octetstring 15
offsets 11

P

partial response 11
path definition 17, 19, 23
pointers 10, 11

R

registration 9, 33, 49
requests 9, 10, 28, 72
response 8
row 5, 20, 21, 44, 72

S

scope keywords 14
security 10, 11, 27, 53
Service Layer 5, 9, 12, 30, 53, 72
service primitives 6
service provider 6
service user 6
set commands 9, 42, 55
specific keyword 23, 39
status codes 56
storage statement 23, 39
string 15, 30, 40, 42

T

table templates 23
tables 5, 20, 24, 44
type statement 22

U

Unicode 14, 16, 18, 53, 72
unregistration 33, 34, 51
unsupported keyword 21, 23, 25

V

value statement 20, 23, 25
versions 26, 28, 29

W

white space 14, 16