Technical Standard

Systems Management:
Common Manageability Programming Interface (CMPI)
Issue 2.0

Technical Standard

Systems Management: Common Manageability Programming Interface (CMPI), Issue 2.0

ISBN: 1-931624-64-X
Document Number: C061

Published by The Open Group, October 2006.

Comments relating to the material contained in this document may be submitted to:

The Open Group
Thames Tower
37-45 Station Road
Reading
Berkshire, RG1 1LX
United Kingdom

or by electronic mail to:

ogspecs@opengroup.org

Contents

Preface

The Open Group

The Open Group is a vendor-neutral and technology-neutral consortium, whose vision of Boundaryless Information Flow will enable access to integrated information within and between enterprises based on open standards and global interoperability. The Open Group works with customers, suppliers, consortia, and other standards bodies. Its role is to capture, understand, and address current and emerging requirements, establish policies, and share best practices; to facilitate interoperability, develop consensus, and evolve and integrate specifications and Open Source technologies; to offer a comprehensive set of services to enhance the operational efficiency of consortia; and to operate the industry's premier certification service, including UNIX certification.

Further information on The Open Group is available at www.opengroup.org.

The Open Group has over 15 years' experience in developing and operating certification programs and has extensive experience developing and facilitating industry adoption of test suites used to validate conformance to an open standard or specification.

More information is available at www.opengroup.org/testing.

The Open Group publishes a wide range of technical documentation, the main part of which is focused on development of Technical and Product Standards and Guides, but which also includes white papers, technical studies, branding and testing documentation, and business titles. Full details and a catalog are available at www.opengroup.org/pubs.

As with all *live* documents, Technical Standards and Specifications require revision to align with new developments and associated international standards. To distinguish between revised specifications which are fully backwards-compatible and those which are not:

- A new Version indicates there is no change to the definitive information contained in the previous publication of that title, but additions/extensions are included. As such, it replaces the previous publication.
- A new Issue indicates there is substantive change to the definitive information contained in the previous publication of that title, and there may also be additions/extensions. As such, both previous and new documents are maintained as current publications.

Readers should note that Corrigenda may apply to any publication. Corrigenda information is published at www.opengroup.org/corrigenda.

This Document

This document is The Open Group Technical Standard for the Common Manageability Programming Interface (CMPI). It has been developed and approved by The Open Group.

Typographical Conventions

All functions, data types, structures, return values, syntax, and code examples are shown in `fixed width font`.

Trademarks

Boundaryless Information Flow™ and TOGAF™ are trademarks and UNIX® and The Open Group® are registered trademarks of The Open Group in the United States and other countries.

The Open Group acknowledges that there may be other brand, company, and product names used in this document that may be covered by trademark protection and advises the reader to verify them independently.

Acknowledgements

The Open Group gratefully acknowledges the contribution of the following people in the development of this Technical Standard:

Guru Bhat	Oracle Corporation
Mark Hamzy	IBM Corporation
Andreas Maier	IBM Corporation
James Marshall	WBEM Solutions, Inc.
Viktor Mihajlovski	IBM Corporation, Linux Technology Centre
Alexander Orr	WBEM Solutions, Inc.
Rick Ratta	Sun Microsystems, Inc.
Adrian Schuur	IBM Corporation, Linux Technology Centre
David Sudlik	IBM Corporation

Referenced Documents

The following documents are referenced in this Technical Standard:

- ANSI/IEEE Std 754-1985, Standard for Binary Floating-Point Arithmetic
- CIM Operations over HTTP v1.1 (DSP0200), Desktop Management Task Force
- CIM v2.2 Specification (DSP0004), Desktop Management Task Force
- CIM XML v2 Specification, Desktop Management Task Force
- CMPI 1.0, Technical Corrigendum 1, available at:
  www.opengroup.org/bookstore/catalog/u064.htm.
- CMPI 1.0, Technical Corrigendum 2, available at:
  www.opengroup.org/bookstore/catalog/u066.htm

1 Introduction

1.1 Purpose

The purpose of this document is to define a C-based programming interface to be used by management servers and their resource-specific extensions. WBEM-based management servers (e.g., WBEM Solutions C CIMOM and OpenPegasus) and management servers based on other concepts (e.g., Resource Management and Control (RMC) of IBM's AIX operating system) all have their own extension interface definitions and terminology.

This document defines a common C-based resource extension interface. With this definition, resource extensions can be re-used in any management server environment supporting this interface.

In order to cover a wide range of management server concepts, this document introduces a new, more neutral terminology. The term Management Broker (MB) is used to define a management server. This is equivalent to various CIM Object Manager implementations or RMC. The term Management Instrumentation (MI) is used to describe management server extensions interfacing directly to the resources being managed. This is equivalent to CIM providers or Resource Managers for RMC.

1.2 Intended Audience

This document is not a programming guide, although it is recognized that one is needed eventually. The intended audience should be knowledgeable in CIM terms and concepts in general, and have knowledge of one or more CIMOM implementations.

1.3 Summary of Characteristics

MIs using this interface definition will have the following characteristics:

- Reduces the complexity of writing MIs. Native MB APIs have been reduced to a smaller, problem-oriented set of functions calls. Automatic garbage collection for CMPI entities further reduces complexity.
- Complete encapsulation and independence of data structures, classes, and implementation language used by the respective MBs. MI is completely isolated from MB and can therefore be re-used in any CMPI supporting MB environment, such as OpenPegasus, WBEM Solutions C Server, and others.
- Uses ANSI C-conforming bindings, thus ensuring platform independence.
- Supports multiple MI packages in one DLL or shared library.
- No MB-specific library is needed – the C header files defined by this Technical Standard are all that is needed to generate CMPI-conforming MIs.

- API definitions fully support local/remote transparency. Assuming suitable proxy processing, CMPI-style MIs can be used as remote MIs without modification.

1.4 Changes Since CMPI 1.0

The following changes have been made since the CMPI 1.0 specification:

- Standardized header file names are now part of the specification: `cmpidt.h`, `cmpift.h`, `cmpios.h`, `cmpipl.h`. These header files include conditional compiler directives for many operating systems. Header files are provided on The Open Group website as a convenience for developers.
- Standardized CMPI macro header (which was originally removed from the CMPI 1.0 specification via Technical Corrigendum 1), `cmpimacs.h`..
- Introduction of an enhanced messaging API in the `CMPIBrokerEncFT` interface: `openMessageFile()`, `closeMessageFile()`, `getMessage2()` (see Section 6.1).
- `setPropertyWithOrigin()` allows providers to set the origin on an instance property, rather than relying on the Management Broker.
- `CMPIInstanceFT::setObjectPath()` may be used to set the keys of an instance.
- Detailed explanation of use of the `CMPISelectExp`, `CMPISelectCond`, `CMPISubCond`, and `CMPIPredOp` structures in query parsing. Addition of COD/DOC Query Normalization (see Appendix A).
- Introduction of the `CMPIError`, `CMPIErrorSrcFormat` structure (see Sections 4.10, 4.11, 4.12, 4.13, and 6.13).
- `CMPIBrokerExtFT` operating system abstraction interface is now required.
- Introduction of the optional interface `CMPIBrokerMemFT`, addition of memory functions (see Section 8.10).
- `CMPICount` replaces `unsigned int` in function return values and index-based parameters (see Section 4.5.6).
- Many typographical errors and inconsistencies with the headers have been corrected.
- Changes incorporated from Technical Corrigendum 1, available at: www.opengroup.org/bookstore/catalog/u064.htm.
- Changes incorporated from Technical Corrigendum 2, available at: www.opengroup.org/bookstore/catalog/u066.htm

2 CMPI Interface General Structure

The environment provided by Management Brokers (MBs) in which Management Instrumentation (MI) executes has to provide the following support:

1. Access to data type manipulation functions

   This support is made accessible to the MI via vectors of entry points. MBs must provide these vectors and provide MB-specific implementation for this support. All defined functions must be implemented.

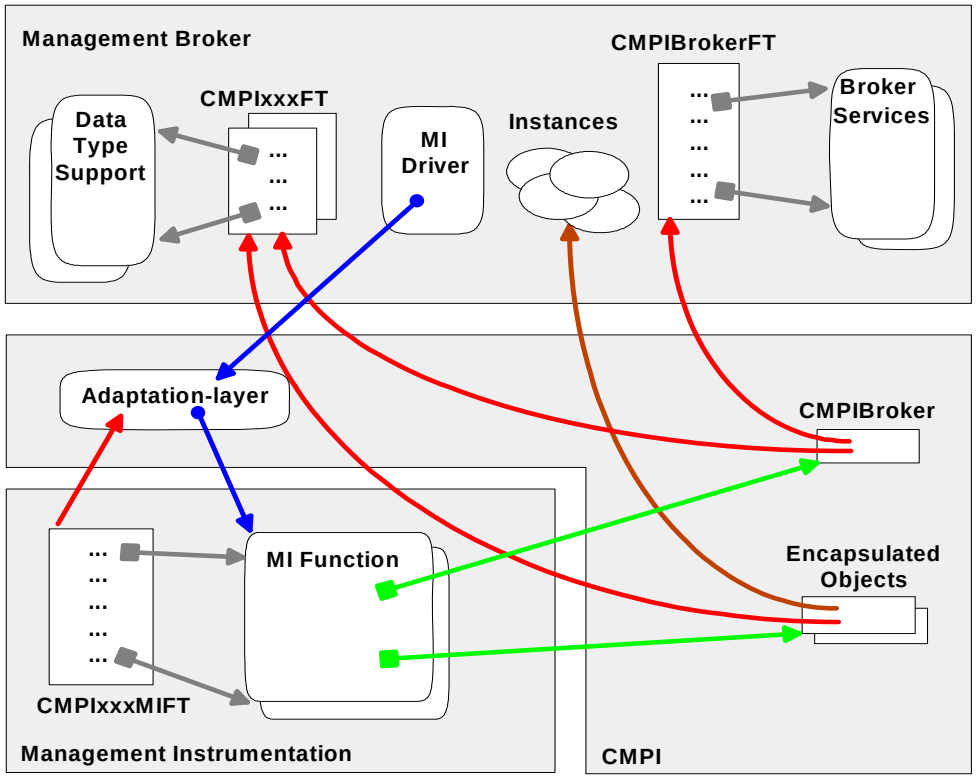
2. Access to MB services

   This support is made accessible to MIs via a vector of entry points. MBs must provide this vector and provide MB-specific implementation. MBs may opt to implement a subset of these services according to their classification (see Chapter 8).

MIs must provide the following:

1. Access to the MI function entry points

   Access is provided by passing a vector of entry points to the MB at initial invocation time. In addition, this vector will have information about the groups of MI functions supported (instance, associations, etc.) and the level of API used.



2. Layout of the vectors and function signatures will be described by a set of C header files. A set of C macros will be provided to hide the vector indirection. For C++ environments, methods will be defined that enable C++-style processing of CMPI entities.

3 API Concepts

In this chapter the principles of CMPI are explained. Code fragments must be seen in the context of the text describing them, and are neither complete nor generic descriptions. The notation "`//..`" indicates that lines of code are left out.

3.1 Highlights

3.1.1 Guiding Principles

There are four basic thoughts that have governed the design of this interface:

1. Existing Management Brokers (MBs) must be able to use CMPI without the need for extensive rework. This resulted in the concept of encapsulation.
2. The external API signatures should be limited in number and, where possible, should be consistent with one other. Ease-of-use is achieved by defining a set of convenience macros.
3. Since most functions result in transformations to the respective MB APIs, CMPI should not introduce new structures containing function parameter data; instead, data should be passed directly as function parameters.
4. CMPI strives to make Management Instrumentation (MI) programming simpler and will eliminate, to the degree possible, intermediate objects such as `CIMValue` and `CIMProperty`.

3.1.2 Encapsulation

Any CMPI data type that has MB implementation-dependent details or contains CMPI data areas is called an "encapsulated data type". An encapsulated data type instance has two parts: a pointer to its MB-dependent implementation and a pointer to the function table giving access to all operations supported by this type. The operation repertoire of encapsulated data types is a major portion of the CMPI interface. This repertoire includes, at a minimum, lifecycle operations (release and clone).

3.1.3 Data Transfer

Most of the API functions deal with transferring property values to and from MBs. Generically there are at least two components needed to define data items: the value itself and its type. Since CIM also supports the notion of NULL values – meaning a property has no value set – a third component is used to define the state of a value indicating whether or not a value has been set.

Values are represented as a union of all types supported by CMPI (`CMPIValue`). Types are represented as a typedef (`CMPIType`). Value states are also represented as a typedef (`CMPIValueState`).

Simple Data Items

For property data transfer calls into CMPI two parameters are used: the address of the value (`CMPIValue*`) and its type (`CMPIType`). If the type does not conform to the schema-defined type, the MB will attempt to convert to the schema-defined type where possible.

Assume a schema-defined property contains a 32-bit signed integer. If the MI wants to set a value, it will pass the address of a `CMPIValue` item containing the value and a `CMPIType` of `CMPI_sint32`. If the MI wants to set a NULL value, it will pass a NULL `CMPIValue` address and a `CMPIType` of `CMPI_sint32`.

Property data transfers from CMPI functions to the MI return three components: the value, its state, and its type as defined by the schema. All three components are bundled into one structure (`CMPIData`).

Assume a schema-defined property contains a 32-bit signed integer. If the value is set, the MB uses the `CMPIData` structure to return the value and the type is set to `CMPI_sint32`. If the value is not set, the MB uses the `CMPIData` structure to return the type set to `CMPI_sint32`, with its state set to `CMPI_nullValue`. In this case, the value union contains no meaningful data.

Array Data Items

Because NULL values must be supported, CMPI cannot directly support C-style arrays. Instead, arrays must be represented as if they are arrays of `CMPIData` structures. CMPI arrays are encapsulated data types (`CMPIArray`).

Before transferring arrays into CMPI the MI first has to create a `CMPIArray` of a `CMPIType`-defined array type (any of `CMPI_xxxA`) and fill it with data using the `setElementAt()` function provided by the `CMPIArray` encapsulated object using the simple data transfer scheme described above.

For property data transfer calls of completed arrays into CMPI the same scheme is used as for transferring simple items; the two parameters are the address of the `CMPIValue` union containing the pointer to the `CMPIArray` and `CMPI_xxxA` as a type.

Property operations returning arrays to an MI use the scheme as used for simple items. If the array property is set, then `CMPIValue` holds a pointer to a `CMPIArray` encapsulated data type with any of the `CMPI_xxxA` types. Otherwise, a `CMPI_nullValue` state is returned along with the schema-defined type (any of `CMPI_xxxA`).

Access to the individual elements of the array is done using the `getElementAt()` function provided by the `CMPIArray` encapsulated data type following the simple data item transfer scheme.

3.1.4 Error Indication

All MI and CMPI functions return a `CMPIStatus` structure, either as a return value or via an output parameter, which contains a `CMPIrc` return code and an optional message in `CMPIString` format. The return codes are compatible with DMTF's CIM error codes. Special codes for additional error indications from CMPI data manipulation routines have been added.

If the `CMPIStatus` output parameter equals NULL, no status structure is returned.

CMPI 2.0 adds the ability for the MI to return more detailed information about an error that occurred. This is done via the `CMPIError` structure.

If the MB supports extended errors (`CMPIBroker.brokerCapabilities` has `CMPI_MB_Supports_Extended_Error` set), then the MI can use this mechanism to return more pertinent error information to the MB.

3.1.5 Threading

CMPI imposes no limits on threading concepts and is itself thread-safe. How MIs themselves are thread-safe and can maintain data integrity is a different story. This depends entirely on the nature of the MI and the resources it controls or represents and on the semantics, or lack thereof, imposed by CIM. MIs must ensure thread-safety, if needed, by using serialization techniques internally.

Support for security context is provided. If an MI wishes to use threads in which CMPI functions are invoked, the thread has to be introduced to CMPI prior to invoking any CMPI functions using the `prepareAttachThread()`, `attachThread()`, and `detachThread()` broker functions. The `prepareAttachThread()` function requires the original `CMPIContext` object to be passed as a parameter and produces a new `CMPIContext` to be used by `attachThread()`.

CMPI defines an abstraction of a subset of threading support functions. This by no means covers all possible threading services made available by various operating systems. Instead, this support would suffice to start and monitor threads used in indication providers. With this subset, CMPI indication providers can be written that are source-portable across MB implementations residing on different operating systems.

3.1.6 Memory Ownership

Due to its encapsulating nature, all complex data structures are created by or via CMPI functions and should therefore be released by or via CMPI. This enables the insulation of MIs from specific memory management techniques used by the MB.

CMPI performs automatic release of all its encapsulated objects used and/or created during an MI function invocation cycle, except for those structures that have been copied explicitly by the MI using the `clone()` function.

Structures that have been copied must eventually be released by the MI explicitly using the `release()` function. This function can be used by long-running indication threads or during invocation cycles to indicate that particular encapsulated objects (not necessarily cloned) are no longer needed.

3.1.7 Implementation of the CMPI API

Implementation of the CMPI API is mostly an adaptation layer that transforms CMPI semantics to the semantics employed by the respective MB. In general, this is a straightforward process; however, there are cases where this is not so easy. Management applications that wish to call CMPI-style MI without using an MB must provide support for all CMPI encapsulated data types. There is an opportunity for a reference implementation for this purpose, but that is outside the scope of this document.

3.1.8 Local/Remote Transparency

This specification describes how MI shall be invoked and how complete encapsulation and isolation is achieved. It is entirely possible to write a proxy mechanism that enables remote or out-of-process invocation and execution of MI. Such environments can be completely transparent for both the MB as well as MI.

3.2 Broker Services

The MB offers services to invoke complex operations similar to operations issued by a `CIMClient`. In addition, the MB gives access to factory services for creating encapsulated types.

A `CMPIBroker` data type is defined to provide access to the broker services. The format of the `CMPIBroker` data type is as follows:

```
typedef struct _CMPIBroker {
    const void* hdl;
    const CMPIBrokerFT* bft;
    const CMPIBrokerEncFT* eft;
    const CMPIBrokerExtFT* xft;
} CMPIBroker;
```

The `CMPIBroker` object is passed as a parameter to the MI factory (`<mi-name>_Create_<mi-type>MI`) at initialization time. The MI will cache this pointer across calls and use it where appropriate.

3.3 Encapsulated Types

A function table is defined for each encapsulated type. Additionally, for each type, a mandatory repertoire of functions is defined and must be implemented by the MB. The sum of all routines accessible via the function tables constitutes the CMPI interface for that MB. This concept enables MIs to work transparently (and simultaneously if needed) with data type instances coming from different MBs. Generically, an encapsulated type has the following format:

```
typedef struct _<E-Type> {
    const void* hdl;
    <E-Type>FT* ft;
} <E-Type>;
```

The following is a partial example of the type and function table for `CMPIInstance`:

```
typedef struct _CMPIInstance {
    const void* hdl;
    const CMPIInstanceFT* ft;
} CMPIInstance;

typedef struct _CMPIInstanceFT {
    const int ftVersion;
    CMPIStatus (*release)(CMPIInstance*);
    CMPIInstance* (*clone)(const CMPIInstance*,CMPIStatus*);
    CMPIData (*getProperty)(const CMPIInstance*,
```

```
        const char*,CMPIStatus*);
    //..
    CMPIStatus (*setProperty)(const CMPIInstance*,const char*,
        const CMPIValue*,CMPIType);
    //..
} CMPIInstanceFT;
```

The `release()` and `clone()` functions are common across and mandatory for every encapsulated object. The `setProperty()` and `getProperty()` functions are typical examples for setting and getting the values in/from encapsulated objects. The `setProperty()`-like operations always have a `CMPIValue` and `CMPIType` pair as parameters. The `getProperty()`-like operations always return a `CMPIData` structure containing the actual value and its type and state. See Chapter 4 for exact definitions of `CMPIValue`, `CMPIValueState`, and `CMPIType`.

```
typedef struct _CMPIData {
    CMPIType type;
    CMPIValueState state;
    CMPIValue value;
} CMPIData;
```

Arrays are returned as `CMPIArray` encapsulated types. This type has functions to get data pertaining to arrays like type, size, and individual elements.

The following example shows some of the actual CMPI functions related to the `CMPIInstance` data type:

```
void
aUselessRoutine(CMPIInstance* ci)
{
    CMPIInstance* cic;
    CMPIArray* ar;
    CMPICount count;
    CMPIStatus rc;
    CMPIValue value;
    int intVal;
    int i;

    value.sint32 = 25;
    cic = ci->ft->clone (ci, &rc);
    rc = cic->ft->setProperty (cic, "prop1", &value,
        CMPI_sint32);
    intVal = cic->ft->getProperty (cic, "prop1",
        &rc).value.sint32;
    ar = cic->ft->getProperty (cic, "propA", &rc).value.array;
    count = ar->ft->getSize (ar, &rc);
    for (i = 0; i < count; i++)
        intVal = ar->ft->getElementAt (ar, i,
            &rc).value.sint32;
}
```

This example shows, among other things, how property values are set and requested. With the exception of `CMPI_chars` data types, set operations always require the use of a pointer

(third parameter) to the value to be set. Since `CMPIValue` is a union of all CMPI data types, a cast operation can be used to point to the value directly:

```
int intVal=25;
rc=cic->ft->setProperty(cic,"prop1",
    (CMPIValue*)&intVal,CMPI_sint32);
```

This technique lends itself very well when macros are used as syntactic sugar:

```
#define CMSetProperty(o,v,t) \
((o)->ft->setProperty((o),(CMPIValue*)(v),(t)))
```

Using this macro, the example above will look like this:

```
rc=CMSetProperty(cic,"prop1",&intVal,CMPI_sint32);
```

When setting `CMPI_chars` data types, the third parameter must be a pointer to the `char` array, instead of a pointer to a pointer to a `char` array.

```
char* str="a char string";
rc=cic->ft->setProperty(cic,"prop3",
    (CMPIValue*)str,CMPI_chars);
```

Or:

```
rc=cic->ft->setProperty(cic,"prop3",
    (CMPIValue*)"a char string",CMPI_chars);
```

Or when using the macro defined above:

```
rc=CMSetProperty(cic,"prop3",str,CMPI_chars);
rc=CMSetProperty(cic,"prop3","a char string",CMPI_chars);
```

3.4 Query and Indication Filtering

This section explains how CMPI deals with filter definitions used for indications and `execQuery`. The goal is to be independent of the language used to express a filter. Disjunctive Normal Form (DNF) is used to represent a filter. WQL/CQL or filter expressions used in LDAP can all be transformed into DNF. This specification requires filter expressions to be presented in DNF at the request of MIs.

The filtering concept for indication support is based on experiences with Solaris WBEM Services and The Open Group OpenPegasus implementations. In essence the filter is representing the complete select expression as a combination of a list defining the projection and a parse tree defining the select condition. A select expression is encapsulated by `CMPISelectExp`. Similarly, for query support, a string containing a select condition is passed to the `execQuery()` function.

3.4.1 Indication Filtering

For indication filtering there are two distinct reasons for having access to the select condition. The most obvious one is for filtering indication candidates. This is probably best done by the MB itself, and presumably the filter structure is set up for effective filter processing.

The other reason is to enable an MI supporting indications to inspect and understand the filter and instrument itself for effective monitoring of the resources it controls. For example, an MI that monitors the availability of a critical service or daemon in a system could arbitrarily poll all services in a system regularly, or intelligently use instrumentation in a system that natively monitors the availability of services by specifying which service(s) should be monitored.

3.4.2 Query

Effective query processing has similar needs: the query process might transform the query into a native format known by the domain this MI function represents. The `execQuery()` function has one more difficulty: it has to parse and query the string which is not necessarily WQL/CQL. In case a WQL/CQL-style language is used, a factory function is offered to transform a query string into a `CMPISelectExp`.

3.4.3 Normalized Select Conditions

In order to inspect a select expression, MIs can request the expression to be transformed into either a conjunction of disjunctions (CoD), or a disjunction of conjunctions (DoC) canonical form, encapsulated as `CMPISelectCond`. In both forms all parenthesis and negations are resolved. As part of this function the projection list can also be requested.

The `CMPISelectCond` is a list of `CMPISubCond` objects. A `CMPISubCond` object is a list of `CMPIPredicates`. Depending on the requests (DoC or CoD), the resulting predicates of a subexpression list are either conjunctive (to be AND'ed) for DoC requests, or disjunctive (to be OR'ed) for CoD requests. The resulting expressions of a normalized select condition are either disjunctive (to be OR'ed) for DoC requests, or conjunctive (to be AND'ed) for CoD requests.

4 Data Types

The CMPI data types are the C data types that Management Instrumentations (MIs) should use in their code in order to encapsulate the specifics of the C data type. They are used in the data manipulation functions, in Management Broker (MB) services, and in the MI functions.

The following header files define the CMPI data types discussed in the following section:

`cmpift.h`	Defines the various CMPI structures and function tables; all providers should include this header.
`cmpidt.h`	Defines CMPI data types; is included and used by `cmpift.h`.
`cmpios.h`	OS-specific defines (for threading structures, etc.); is included by `cmpift.h`.
`cmpipl.h`	Platform checker. This header verifies that a valid CMPI_PLATFORM has been specified; is included by `cmpidt.h`.
`cmpimacs.h`	CMPI macro functions. A set of macro functions to ease the use of the various function tables; is included by `cmpift.h`.

For the benefit of developers, copies of these header files will be downloadable from the CMPI website (www.opengroup.org/tech/management/cmpi). Please note that these files are provided as a convenience only. In the case of any discrepancy between the header files and this Technical Standard (incorporating any subsequent Technical Corrigenda), this Technical Standard shall be definitive.

4.1 CMPI Encapsulated Data Types

A major requirement for CMPI is to hide implementation details of MB-specific data structures. CMPI uses pointers to opaque structures containing the MB-specific implementations, including a function pointer table providing type-specific support. This table, among others, contains MB-specific memory management support functions. In this document, the symbol `<E-Type>` is used to generically refer to the encapsulated types listed in the following table:

Table 1: Encapsulated Data Types

CMPI Data Type <E-Type>	C Data Type
`CMPIInstance`	`struct _CMPIInstance`
`CMPIObjectPath`	`struct _CMPIObjectPath`
`CMPIArgs`	`struct _CMPIArgs`
`CMPIEnumeration`	`struct _CMPIEnumeration`
`CMPIArray`	`struct _CMPIArray`

CMPI Data Type <E-Type>	C Data Type
CMPISelectExp	struct _CMPISelectExp
CMPISelectCond	struct _CMPISelectCond
CMPISubCond	struct _CMPISubCond
CMPIPredicate	struct _CMPIPredicate
CMPIDateTime	struct _CMPIDateTime
CMPIContext	struct _CMPIContext
CMPIResult	struct _CMPIResult
CMPIErrror	struct _CMPIError

```
struct _CMPIInstance;
struct _CMPIObjectPath;
struct _CMPIArgs;
struct _CMPIEnumeration;
struct _CMPIArray;
struct _CMPISelectExp;
struct _CMPISelectCond;
struct _CMPISubCond;
struct _CMPIPredicate;
struct _CMPIDateTime;
struct _CMPIContext;
struct _CMPIResult;
struct _CMPIError;
typedef struct _CMPIInstance    CMPIInstance;
typedef struct _CMPIObjectPath  CMPIObjectPath;
typedef struct _CMPIArgs        CMPIArgs;
typedef struct _CMPIEnumeration CMPIEnumeration;
typedef struct _CMPIArray       CMPIArray;
typedef struct _CMPISelectExp   CMPISelectExp;
typedef struct _CMPISelectCond  CMPISelectCond;
typedef struct _CMPISubCond     CMPISubCond;
typedef struct _CMPIPredicate   CMPIPredicate;
typedef struct _CMPIContext     CMPIContext;
typedef struct _CMPIResult      CMPIResult;
typedef struct _CMPIDateTime    CMPIDateTime;
typedef struct _CMPIError       CMPIError;
```

4.2 CMPI String Data

For string data, two formats are supported, depending on usage: input to CMPI functions is as ordinary zero-terminated UTF-8 character arrays, and strings generated by CMPI functions are returned as pointers to `CMPIString` structures.

The reason for differentiating here is a matter of convenience and technical necessity:

1. Specifying a property name as input to a function using `CMPIString` would require using a `CMPIString` factory without any visible and/or measurable advantage.

2. Strings being returned from CMPI are dynamic in nature and minimally require memory management support. Therefore, they follow the encapsulation concept.

Gaining access to the underlying character array can be achieved by calling `str->ft->getCharPtr` or using the `CMGetChars` macro. The underlying character array cannot be modified directly.

Table 2: String Data Types

CMPI Data Type	C Data Type
`CMPIString`	`struct _CMPIString`

```
struct _CMPIString;
typedef struct _CMPIString CMPIString;
```

4.3 CMPI Simple Data Types

Most of the CMPI data type manipulation functions use as input and return simple data items such as integers, booleans, and floating-point numbers. All simple CIM types are supported via a convenience layer. The following is a mapping of simple CIM data types to the corresponding simple CMPI data types.

Table 3: Simple CMPI Data Types

Simple CIM Data Type	CMPI Data Type <C-Type>	C Data Type
`boolean`	`CMPIBoolean`	unsigned char (0: false, any other value: true)
`char16`	`CMPIChar16`	16-bit unsigned integer
`uint8`	`CMPIUint8`	8-bit unsigned integer
`uint16`	`CMPIUint16`	16-bit unsigned integer
`uint32`	`CMPIUint32`	32-bit unsigned integer
`uint64`	`CMPIUint64`	64-bit unsigned integer
`sint8`	`CMPISint8`	8-bit integer
`sint16`	`CMPISint16`	16-bit integer
`sint32`	`CMPISint32`	32-bit integer
`sint64`	`CMPISint64`	64-bit integer
`real32`	`CMPIReal32`	32-bit floating-point
`real64`	`CMPIReal64`	64-bit floating-point
`string`	`CMPIString`[1] `char*`	Encapsulated string type Zero-terminated UTF-8 char array

1 For this discussion, `CMPIString` is considered to be a simple CIM data type, although it is an encapsulated type. See Section 4.2 for declaration and rationale.

Simple CIM Data Type	CMPI Data Type <C-Type>	C Data Type
datetime	CMPIDateTime[2]	Date/time structure

```
typedef unsigned char           CMPIBoolean;
typedef unsigned short          CMPIChar16;
typedef unsigned char           CMPIUint8;
typedef unsigned short          CMPIUint16;
typedef unsigned int            CMPIUint32;
typedef unsigned long long      CMPIUint64;[3]
typedef signed char             CMPISint8;
typedef short                   CMPISint16;
typedef signed int              CMPISint32;
typedef long long               CMPISint64;[4]
typedef float                   CMPIReal32;
typedef double                  CMPIReal64;
```

4.4 CMPI Miscellaneous Data Types

CMPIValuePtr is used for context data only. It is used to describe raw unformatted data areas. It points to a data area and contains the length of the area.

Table 4: Miscellaneous Data Types

CMPI Data Type	C Data Type
CMPIValuePtr	struct _CMPIValuePtr

```
typedef struct _CMPIValuePtr {
    void* ptr;
    CMPICount length;
} CMPIValuePtr;
```

4.5 CMPI Types and Values

Three components are used to define value, state, and type (CMPIValue, CMPIValueState, and CMPIType). The value and type components are used by MIs when transferring property data to the MB (set operations). All three components are returned to an MI by the MB when accessing properties; in which case they are bundled in a structure called CMPIData.

4.5.1 CMPIData

CMPIData is a structure that holds all three components returned to an MI when accessing property data using functions like getProperty().

```
typedef struct _CMPIData {
```

2 CMPIDateTime is implemented as an encapsulated data type.
3 On Windows, this is typedef unsigned __int64 CMPIUint64.
4 On Windows, this is typedef __int 64 CMPISint64.

```
    CMPIType type;
    CMPIValueState state;
    CMPIValue value;
} CMPIData;
```

The `type` parameter is set to reflect the property type as defined in the schema, the `state` parameter indicates whether a value is set (not a NULL value) or originates from a `CMPIObjectPath` key-value, and the `value` parameter contains the actual value if not a NULL value.

4.5.2 CMPIType

A type discriminator (`CMPIType`) is used to define data types of corresponding values. It is used either as part of a `CMPIData` structure, or as an additional parameter when setting property values.

```
typedef unsigned short CMPIType;

#define CMPI_null         (0)

#define CMPI_SIMPLE       (2)
#define CMPI_boolean      (2+0)
#define CMPI_char16       (2+1)

#define CMPI_REAL         ((2)<<2)
#define CMPI_real32       ((2+0)<<2)
#define CMPI_real64       ((2+1)<<2)

#define CMPI_UINT         ((8)<<4)
#define CMPI_uint8        ((8+0)<<4)
#define CMPI_uint16       ((8+1)<<4)
#define CMPI_uint32       ((8+2)<<4)
#define CMPI_uint64       ((8+3)<<4)
#define CMPI_SINT         ((8+4)<<4)
#define CMPI_sint8        ((8+4)<<4)
#define CMPI_sint16       ((8+5)<<4)
#define CMPI_sint32       ((8+6)<<4)
#define CMPI_sint64       ((8+7)<<4)
#define CMPI_INTEGER      ((CMPI_UINT | CMPI_SINT))

#define CMPI_ENC          ((16)<<8)
#define CMPI_instance     ((16+0)<<8)
#define CMPI_ref          ((16+1)<<8)
#define CMPI_args         ((16+2)<<8)
#define CMPI_class        ((16+3)<<8)
#define CMPI_filter       ((16+4)<<8)
#define CMPI_enumeration  ((16+5)<<8)
#define CMPI_string       ((16+6)<<8)
#define CMPI_chars        ((16+7)<<8)
#define CMPI_dateTime     ((16+8)<<8)
#define CMPI_ptr          ((16+9)<<8)
#define CMPI_charsptr     ((16+10)<<8)

#define CMPI_ARRAY        ((1)<<13)
```

```
#define CMPI_SIMPLEA      (CMPI_ARRAY | CMPI_SIMPLE)
#define CMPI_booleanA     (CMPI_ARRAY | CMPI_boolean)
#define CMPI_char16A      (CMPI_ARRAY | CMPI_char16)

#define CMPI_REALA        (CMPI_ARRAY | CMPI_REAL)
#define CMPI_real32A      (CMPI_ARRAY | CMPI_real32)
#define CMPI_real64A      (CMPI_ARRAY | CMPI_real64)

#define CMPI_UINTA        (CMPI_ARRAY | CMPI_UINT)
#define CMPI_uint8A       (CMPI_ARRAY | CMPI_uint8)
#define CMPI_uint16A      (CMPI_ARRAY | CMPI_uint16)
#define CMPI_uint32A      (CMPI_ARRAY | CMPI_uint32)
#define CMPI_uint64A      (CMPI_ARRAY | CMPI_uint64)
#define CMPI_SINTA        (CMPI_ARRAY | CMPI_SINT)
#define CMPI_sint8A       (CMPI_ARRAY | CMPI_sint8)
#define CMPI_sint16A      (CMPI_ARRAY | CMPI_sint16)
#define CMPI_sint32A      (CMPI_ARRAY | CMPI_sint32)
#define CMPI_sint64A      (CMPI_ARRAY | CMPI_sint64)
#define CMPI_INTEGERA     (CMPI_ARRAY | CMPI_INTEGER)

#define CMPI_ENCA         (CMPI_ARRAY | CMPI_ENC)
#define CMPI_stringA      (CMPI_ARRAY | CMPI_string)
#define CMPI_charsA       (CMPI_ARRAY | CMPI_chars)
#define CMPI_dateTimeA    (CMPI_ARRAY | CMPI_dateTime)
#define CMPI_instanceA    (CMPI_ARRAY | CMPI_instance)
#define CMPI_refA         (CMPI_ARRAY | CMPI_ref)
#define CMPI_charsptrA    (CMPI_ARRAY | CMPI_charsptr)

// The following are CMPIObjectPath key-type synonyms
// and are valid only when CMPI_keyValue of
// CMPIValueState is set.

#define CMPI_keyInteger   (CMPI_sint64)
#define CMPI_keyString    (CMPI_string)
#define CMPI_keyBoolean   (CMPI_boolean)
#define CMPI_keyRef       (CMPI_ref)

// The following are predicate types only.

#define CMPI_charString      (CMPI_string)
#define CMPI_realString      (CMPI_string | CMPI_real64)
#define CMPI_integerString   (CMPI_string | CMPI_integer64)
#define CMPI_numericString   (CMPI_string | CMPI_sint64 |
                              CMPI_real64)
#define CMPI_booleanString   (CMPI_string | CMPI_boolean)
#define CMPI_dateTimeString  (CMPI_string | CMPI_dateTime)
#define CMPI_classNameString (CMPI_string | CMPI_class)
#define CMPI_nameString      (CMPI_string | ((16+10)<<8)
```

Note that `CMPI_null` is returned by CMPI only in case of error. It is also used by the MI with property setting operations to indicate a NULL value being set.

4.5.3 CMPIValueState

`CMPIValueState` describes the state of the actual value, whether it is set (not a NULL value) or emanated from a `CMPIObjectPath` key value.

```
typedef unsigned short CMPIValueState;

#define CMPI_goodValue (0)
#define CMPI_nullValue (1<<8)
#define CMPI_keyValue  (2<<8)
#define CMPI_notFound  (4<<8)
#define CMPI_badValue  (0x80<<8)
```

4.5.4 CMPIValue

`CMPIValue` is a union that can hold any of the data types defined in CMPI.

```
typedef union _CMPIValue {
    CMPIBoolean             boolean;
    CMPIChar16              char16;
    CMPIUint8               uint8;
    CMPIUint16              uint16;
    CMPIUint32              uint32;
    CMPIUint64              uint64;
    CMPISint8               sint8;
    CMPISint16              sint16;
    CMPISint32              sint32;
    CMPISint64              sint64;
    CMPIReal32              real32;
    CMPIReal64              real64;

    CMPIInstance*           inst;
    CMPIObjectPath*         ref;
    CMPIArgs*               args;
    CMPISelectExp*          filter;
    CMPIEnumeration*        Enum;
    CMPIArray*              array;
    CMPIString*             string;
    char*                   chars;
    CMPIDateTime*           dateTime;
    CMPIValuePtr            dataPtr;

    CMPISint8               Byte;
    CMPISint16              Short;
    CMPISint32              Int;
    CMPISint64              Long;
    CMPIReal32              Float;
    CMPIReal64              Double;
} CMPIValue;
```

4.5.5 CMPIPredOp

`CMPIPredOP` represents a unary or binary operator, in parsed queries (see Section 6.12 and Appendix A).

```
typedef enum _CMPIPredOp {
    CMPI_PredOp_Equals              = 1,
    CMPI_PredOp_NotEquals           = 2,
    CMPI_PredOp_LessThan            = 3,
    CMPI_PredOp_GreaterThanOrEquals = 4,
    CMPI_PredOp_GreaterThan         = 5,
    CMPI_PredOp_LessThanOrEquals    = 6,
    CMPI_PredOp_Isa                 = 7,
    CMPI_PredOp_NotIsa              = 8,
    CMPI_PredOp_Like                = 9,
    CMPI_PredOp_NotLike             = 10
}CMPIPredOp;
```

4.5.6 CMPICount

`CMPICount` is an unsigned integer that is used as an argument or return type to specify the number of array elements, arguments, etc.

```
typedef unsigned int CMPICount;
```

4.5.7 CMPIMsgFileHandle

`CMPIMsgFileHandle` is a void pointer that is used to refer to an open message file.

```
typedef void* CMPIMsgFileHandle;
```

4.6 Null Value Specification

As mentioned in Section 3.1.3, CMPI supports the notion of NULL values. When CMPI returns a NULL value, the `CMPI_nullValue` flag of `CMPIValueState` is set to true, `CMPIValue` is set to all binary zeros, and `CMPIType` contains a valid type. Two methods of passing a NULL value to CMPI are supported: using a NULL pointer as the `CMPIValue` pointer, or having a NULL pointer in `CMPIValue` for an encapsulated data type pointer. The first method must be used for simple data items and can be used for all encapsulated objects. The second method can only be used for encapsulated objects.

Examples

```
CMPIInstance* ci;
CMPIValue val;

// Returning a simple NULL data item:
ci->ft->setProperty(ci,”propName1”,NULL,CMPI_sint32);

// Returning a NULL array data type:
ci->ft->setProperty(ci,”propName2”,NULL,CMPI_sint32A);

// Returning a pointer to a NULL array data type:
val.array=NULL;
ci->ft->setProperty(ci,”propName2”,&val,CMPI_sint32A);
```

4.7 CMPI Return Codes

The following return codes are recognized:

```
typedef enum _CMPIrc {
    CMPI_RC_OK                              = 0,
    CMPI_RC_ERR_FAILED                      = 1,
    CMPI_RC_ERR_ACCESS_DENIED               = 2,
    CMPI_RC_ERR_INVALID_NAMESPACE           = 3,
    CMPI_RC_ERR_INVALID_PARAMETER           = 4,
    CMPI_RC_ERR_INVALID_CLASS               = 5,
    CMPI_RC_ERR_NOT_FOUND                   = 6,
    CMPI_RC_ERR_NOT_SUPPORTED               = 7,
    CMPI_RC_ERR_CLASS_HAS_CHILDREN          = 8,
    CMPI_RC_ERR_CLASS_HAS_INSTANCES         = 9,
    CMPI_RC_ERR_INVALID_SUPERCLASS          = 10,
    CMPI_RC_ERR_ALREADY_EXISTS              = 11,
    CMPI_RC_ERR_NO_SUCH_PROPERTY            = 12,
    CMPI_RC_ERR_TYPE_MISMATCH               = 13,
    CMPI_RC_ERR_QUERY_LANGUAGE_NOT_SUPPORTED = 14,
    CMPI_RC_ERR_INVALID_QUERY               = 15,
    CMPI_RC_ERR_METHOD_NOT_AVAILABLE        = 16,
    CMPI_RC_ERR_METHOD_NOT_FOUND            = 17,

/* The following return codes are used by cleanup()
   calls only. */

    CMPI_RC_DO_NOT_UNLOAD                   = 50,
    CMPI_RC_NEVER_UNLOAD                    = 51,

/* Internal CMPI return codes */

    CMPI_RC_ERR_INVALID_HANDLE              = 60,
    CMPI_RC_ERR_INVALID_DATA_TYPE           = 61
} CMPIrc;

/* Hosting OS errors */

    CMPI_RC_ERROR_SYSTEM                    = 100,
    CMPI_RC_ERROR                           = 200
} CMPIrc;
```

4.8 CMPI Severity Codes

The following severity codes are recognized:

```
typedef enum _CMPISeverity {
    CMPI_SEV_ERROR   = 1,
    CMPI_SEV_INFO    = 2,
    CMPI_SEV_WARNING = 3,
    CMPI_DEV_DEBUG   = 4
}CMPISeverity;
```

4.9 CMPI Level Codes

The following level codes are recognized:

```
typedef enum _CMPILevel {
    CMPI_LEV_INFO     = 1, /* generic information */
    CMPI_LEV_WARNING  = 2, /* warnings */
    CMPI_LEV_VERBOSE  = 3  /* detailed/specific information */
}CMPILevel;
```

4.10 CMPIError Types

The following error types are supported:

```
typedef enum _CMPIErrorType {
    Unknown = 0,
    Other = 1,
    CommunicationsError = 2,
    QualityOfServiceError = 3,
    SoftwareError = 4,
    HardwareError = 5,
    EnvironmentalError = 6,
    SecurityError = 7,
    OversubscriptionError = 8,
    UnavailableResourceError = 9,
    UnsupportedOperationError = 10,
} CMPIErrorType;
```

4.11 CMPIError Severity Types

The following severity levels are supported:

```
typedef enum _CMPIErrorSeverity {
    Unknown = 0,
    Low = 2,
    Medium = 3,
    High = 4,
    Fatal = 5,
} CMPIErrorSeverity;
```

4.12 CMPIError Probable Cause Codes

The following probable cause codes are supported:

```
typedef enum _CMPIErrorProbableCause {
    Unknown = 0,
    Other = 1,
    Adapter_Card_Error = 2,
    Application_Subsystem_Failure = 3,
    Bandwidth_Reduced = 4,
    Connection_Establishment_Error = 5,
    Communications_Protocol_Error = 6,
```

```
Communications_Subsystem_Failure = 7,
Configuration/Customization_Error = 8,
Congestion = 9,
Corrupt_Data = 10,
CPU_Cycles_Limit_Exceeded = 11,
Dataset/Modem_Error = 12,
Degraded_Signal = 13,
DTE_DCE_Interface_Error = 14,
Enclosure_Door_Open = 15,
Equipment_Malfunction = 16,
Excessive_Vibration = 17,
File_Format_Error = 18,
Fire_Detected = 19,
Flood_Detected = 20,
Framing_Error = 21,
HVAC_Problem = 22,
Humidity_Unacceptable = 23,
IO_Device_Error = 24,
Input_Device_Error = 25,
LAN_Error = 26,
Non_Toxic_Leak_Detected = 27,
Local_Node_Transmission_Error = 28,
Loss_of_Frame = 29,
Loss_of_Signal = 30,
Material_Supply_Exhausted = 31,
Multiplexer_Problem = 32,
Out_of_Memory = 33,
Output_Device_Error = 34,
Performance_Degraded = 35,
Power_Problem = 36,
Pressure_Unacceptable = 37,
Processor_Problem = 38,
Pump_Failure = 39,
Queue_Size_Exceeded = 40,
Receive_Failure = 41,
Receiver_Failure = 42,
Remote_Node_Transmission_Error = 43,
Resource_at_or_Nearing_Capacity = 44,
Response_Time_Excessive = 45,
Retransmission_Rate_Excessive = 46,
Software_Error = 47,
Software_Program_Abnormally_Terminated = 48,
Software_Program_Error = 49,
Storage_Capacity_Problem = 50,
Temperature_Unacceptable = 51,
Threshold_Crossed = 52,
Timing_Problem = 53,
Toxic_Leak_Detected = 54,
Transmit_Failure = 55,
Transmitter_Failure = 56,
Underlying_Resource_Unavailable = 57,
Version_Mismatch = 58,
Previous_Alert_Cleared = 59,
```

```
Login_Attempts_Failed = 60,
Software_Virus_Detected = 61,
Hardware_Security_Breached = 62,
Denial_of_Service_Detected = 63,
Security_Credential_Mismatch = 64,
Unauthorized_Access = 65,
Alarm_Received = 66,
Loss_of_Pointer = 67,
Payload_Mismatch = 68,
Transmission_Error = 69,
Excessive_Error_Rate = 70,
Trace_Problem = 71,
Element_Unavailable = 72,
Element_Missing = 73,
Loss_of_Multi_Frame = 74,
Broadcast_Channel_Failure = 75,
Invalid_Message_Received = 76,
Routing_Failure = 77,
Backplane_Failure = 78,
Identifier_Duplication = 79,
Protection_Path_Failure = 80,
Sync_Loss_or_Mismatch = 81,
Terminal_Problem = 82,
Real_Time_Clock_Failure = 83,
Antenna_Failure = 84,
Battery_Charging_Failure = 85,
Disk_Failure = 86,
Frequency_Hopping_Failure = 87,
Loss_of_Redundancy = 88,
Power_Supply_Failure = 89,
Signal_Quality_Problem = 90,
Battery_Discharging = 91,
Battery_Failure = 92,
Commercial_Power_Problem = 93,
Fan_Failure = 94,
Engine_Failure = 95,
Sensor_Failure = 96,
Fuse_Failure = 97,
Generator_Failure = 98,
Low_Battery = 99,
Low_Fuel = 100,
Low_Water = 101,
Explosive_Gas = 102,
High_Winds = 103,
Ice_Buildup = 104,
Smoke = 105,
Memory_Mismatch = 106,
Out_of_CPU_Cycles = 107,
Software_Environment_Problem = 108,
Software_Download_Failure = 109,
Element_Reinitialized = 110,
Timeout = 111,
Logging_Problems = 112,
```

```
    Leak_Detected_113,
    Protection_Mechanism_Failure = 114,
    Protecting_Resource_Failure = 115,
    Database_Inconsistency = 116,
    Authentication_Failure = 117,
    Breach_of_Confidentiality = 118,
    Cable_Tamper = 119,
    Delayed_Information = 120,
    Duplicate_Information = 121,
    Information_Missing = 122,
    Information_Modification = 123,
    Information_Out_of_Sequence = 124,
    Key_Expired = 125,
    Non_Repudiation_Failure = 126,
    Out_of_Hours_Activity = 127,
    Out_of_Service = 128,
    Procedural_Error = 129,
    Unexpected_Information = 130,
} CMPIErrorProbableCause;
```

4.13 CMPIError Source Format Types

The following formats are supported:

```
typedef _CMPIErrorSrcFormat {
    Unknown = 0,
    Other = 1,
    CIMObjectHandle = 2,
} CMPIErrorSrcFormat;
```

5 MI Function Signatures

Management Instrumentation (MI) as used in this specification corresponds to what is commonly known as a “Provider” in the CIMOM world. In the past, it has been common practice to subdivide functions offered via Providers into optional groups of related functions. CMPI supports five related groups of functions as follows:

- Instance MI
- Association MI
- Property MI
- Method MI
- Indication MI

The related groups are usually linked together in a dynamically loadable library that is known by the Management Broker (MB) using a naming convention. Every group is identifiable by the following pair of names:

```
load-lib-name
mi-name
```

The MB will determine these names either by convention or via “provider registration”. `Load-lib-name` is used by the MB to load a library where it expects to find MI groups defined above.

`Mi-name`, which is normally the same as the schema or class name, is used to locate the initializing functions within the library. This function is called to initialize the individual MI groups and returns the function table for this group.

Notice that all functions of implemented groups *must* be presented as valid entry points (not as a NULL pointer). When there are legitimate reasons not to support a specific function, a CMPI_ERR_NOT_SUPPORTED return code *must* be used to indicate this.

The signatures correspond largely with the similar constructs found in most CIM Object Manager implementations.

All functions return a `CMPIStatus` structure to indicate success or failure. Setting `CMPIStatus.rc` to `CMPI_RC_OK` indicates successful completion.

The term `<mi-type>` is used in a number of template signatures. `<mi-type>` can be substituted by either Instance, Association, Property, Method, or Indication.

5.1 MI Factories

After MIs are loaded by the MB, the MB will attempt to determine the MI function groups supported by the MI and the respective entry points of the individual functions. It does so by verifying the existence of MI factory entry points.

MI factory entry points have one of the following formats:

```
<mi-name>_Create_<mi-type>MI
_Generic_Create_<mi-type>MI
```

Once an MI factory has been found, it will be invoked. The factory has to return a pointer to a valid `CMPI<mi-type>MI`[5] structure, or NULL. The major component of this structure is the function table enabling access to the individual MI group functions. Factories may return a NULL pointer indicating that a particular MI type is not supported, despite the fact that the factory is found.

`CMPI<mi-type>MI` can be extended by MIs for maintaining MI-specific data across MI function invocations. It should be noted, however, that MIs can be unloaded by an MB at any time.

Two types of factories are supported: MI-specific factories and generic factories. Generic factories are used when a single MI acts as a proxy for other MIs. MBs must first verify the existence of generic factories; if found, MI-specific factories are to be ignored.

MI factories also have a cleanup entry point. This function is accessed via the `CMPI<mi-type>MI` structure and is called for every MI group prior to unloading the MI.

Notice that all MI factory functions found are called, meaning a particular MI per supported MI group is asked to perform intialization or cleanup. The MI implementation can optimize to handle this effectively.

5.1.1 MI-Specific Factories

MI-specific factories have the following signature template:

```
CMPI<mi-type>MI* <mi-name>_Create_<mi-type>MI(
    const CMPIBroker*,
    const CMPIContext*, CMPIStatus*);
```

5.1.2 Generic MI Factories

Generic factories differ from MI-specific factories in that `<mi-name>` is passed as a parameter. Generic MI factories have the following signature template:

```
CMPI<mi-type>MI*_Generic_Create_<mi-type>MI(const CMPIBroker*,
    const CMPIContext*,const char*,CMPIStatus*);
```

5.1.3 MI Function Tables

The MI function tables are used by the MB to invoke individual MI functions. MI factories must return a pointer to a `CMPI<mi-type>MI` structure. This structure must point to a fully initialized `CMPI<mi-type>MIFT` structure. Alternatively, MI factories may return a NULL pointer indicating that this function group is not supported.

The `CMPI<mi-type>MIFT` structure will be used by the MB to access the individual MI functions. For this version of the CMPI Technical Standard, the value of `ftVersion` should be 200.

---

5 The `<mi-type>` value can be Instance, Association, Property, Method, or Indication.

`CMPI<mi-type>MIFT.ftVersion` indicates to the MB the function table version used by the MI. The MB may refuse to cooperate with MIs that use table versions not supported by a specific MB.

`CMPI<mi-type>MIFT.miVersion` and `CMPI<mi-type>MIFT.miName` can be used by MI to expose informational details and may or may not be used by the MB.

5.1.3.1 CMPIInstanceMIFT

```
typedef struct _CMPIInstanceMIFT {
    int ftVersion;
    int miVersion;
    char* miName;
    CMPIStatus (*cleanup)
        (CMPIInstanceMI*,const CMPIContext*,CMPIBoolean);
    CMPIStatus (*enumerateInstanceNames)
        (CMPIInstanceMI*,const CMPIContext*,const CMPIResult*,
         const CMPIObjectPath*);
    CMPIStatus (*enumerateInstances)
        (CMPIInstanceMI*,const CMPIContext*,const CMPIResult*,
         const CMPIObjectPath*,const char**);
    CMPIStatus (*getInstance)
        (CMPIInstanceMI*,const CMPIContext*,const CMPIResult*,
         const CMPIObjectPath*,const char**);
    CMPIStatus (*createInstance)
        (CMPIInstanceMI*,const CMPIContext*,const CMPIResult*,
         const CMPIObjectPath*,const CMPIInstance*);
    CMPIStatus (*modifyInstance)
        (CMPIInstanceMI*,const CMPIContext*,const CMPIResult*,
         const CMPIObjectPath*,const CMPIInstance*,
         const char**);
    CMPIStatus (*deleteInstance)
        (CMPIInstanceMI*,const CMPIContext*,const CMPIResult*,
         const CMPIObjectPath*);
    CMPIStatus (*execQuery)
        (CMPIInstanceMI*,const CMPIContext*,const CMPIResult*,
         const CMPIObjectPath*,const char*,const char*);
} CMPIInstanceMIFT;

typdef struct _CMPIInstanceMI {
   const void* hdl;
   struct _CMPIInstanceMIFT* ft;
} CMPIInstanceMI;
```

5.1.3.2 CMPIAssociationMIFT

```
typedef struct _CMPIAssociationMIFT {
    int ftVersion;
    int miVersion;
    char* miName;
    CMPIStatus (*cleanup)
        (CMPIAssociationMI*,const CMPIContext*,CMPIBoolean);
    CMPIStatus (*associators)
        (CMPIAssociationMI*,const CMPIContext*,
         const CMPIResult*,
```

```
             const CMPIObjectPath*,const char*,const char*,
             const char*,const char*,const char**);
        CMPIStatus (*associatorNames)
            (CMPIAssociationMI*,const CMPIContext*,
             const CMPIResult*,
             const CMPIObjectPath*,const char*,const char*,
             const char*,const char*);
        CMPIStatus (*references)
            (CMPIAssociationMI*,const CMPIContext*,
             const CMPIResult*,
             const CMPIObjectPath*,const char*,const char*,
             const char**);
        CMPIStatus (*referenceNames)
            (CMPIAssociationMI*,const CMPIContext*,
             const CMPIResult*,
             const CMPIObjectPath*,const char*,const char*);
    } CMPIAssociationMIFT;

    typdef struct _CMPIAssocationMI {
        const void* hdl;
        struct _CMPIAssociationMIFT* ft;
    } CMPIAssociationMI;
```

5.1.3.3 *CMPIMethodMIFT*

```
    typdef struct _CMPIMethodMIFT {
        int ftVersion;
        int miVersion;
        char* miName;
        CMPIStatus (*cleanup)
            (CMPIMethodMI*,const CMPIContext*,CMPIBoolean);
        CMPIStatus (*invokeMethod)
            (CMPIMethodMI*,const CMPIContext*,const CMPIResult*,
             const CMPIObjectPath*,const char*,
             const CMPIArgs*,CMPIArgs*);
    } CMPIMethodMIFT;

    typdef struct _CMPIMethodMI {
        const void* hdl;
        struct _CMPIMethodMIFT* ft;
    } CMPIMethodMI;
```

5.1.3.4 *CMPIIndicationMIFT*

```
    typedef struct _CMPIIndicationMIFT {
        int ftVersion;
        int miVersion;
        char* miName;
        CMPIStatus (*cleanup)
            (CMPIIndicationMI*,const CMPIContext*,CMPIBoolean);
        CMPIStatus (*authorizeFilter)
            (CMPIIndicationMI*,const CMPIContext*,
             const CMPISelectExp*,const char*,
             const CMPIObjectPath*,const char*);
        CMPIStatus (*mustPoll)
```

```
            (CMPIIndicationMI*,const CMPIContext*,
             const CMPISelectExp*,const char*,
             const CMPIObjectPath*);
        CMPIStatus (*activateFilter)
            (CMPIIndicationMI*,const CMPIContext*,
             const CMPISelectExp*,const char*,
             const CMPIObjectPath*,CMPIBoolean);
        CMPIStatus (*deActivateFilter)
            (CMPIIndicationMI*,const CMPIContext*,
             const CMPISelectExp*,const char*,const
CMPIObjectPath*,
             CMPIBoolean);
        CMPIStatus (*enableIndications)
            (CMPIIndicationMI*,const CMPIContext*);
        CMPIStatus (*disableIndications)
            (CMPIIndicationMI*,const CMPIContext*);
    } CMPIIndicationMIFT;

    typedef struct _CMPIIndicationMI {
        const void* hdl;
        struct _CMPIIndicationMIFT* ft;
    } CMPIIndicationMI;
```

5.1.3.5 *CMPIPropertyMIFT*

```
    typedef struct _CMPIPropertyMIFT {
        int ftVersion;
        int miVersion;
        const char* miName;
        CMPIStatus (*cleanup)
            (CMPIPropertyMI*,const CMPIContext*,
             CMPIBoolean);
        CMPIStatus (*setProperty)
            (CMPIPropertyMI*,const CMPIContext*,
             const CMPIResult*,const CMPIObjectPath*,
             const char*,const CMPIData);
        CMPIStatus (*getProperty)
            (CMPIPropertyMI*, const CMPIContext*,
             const CMPIResult*,
             const CMPIObjectPath*, const char*);
    };
```

5.1.4 MI Factory Functions

The MI factories are accessed by the MB through the following functions:

<mi-name>_Create_<mi-type>MI()

NAME

<mi-name>_Create_InstanceMI()

<mi-name>_Create_AssociationMI()

<mi-name>_Create_PropertyMI()

<mi-name>_Create_MethodMI()

<mi-name>_Create_IndicationMI()

SYNOPSIS

```
CMPI<mi-type>MI* <mi-name>_Create_<mi-type>MI(
    const CMPIBroker* broker,
    const CMPIContext* ctx,
    CMPIStatus* rc);
```

DESCRIPTION

The <mi-name>_Create_<mi-type>MI() functions shall perform any necessary initialization operations of this MI group.

The broker argument is a pointer to a CMPIBroker structure. This structure can be used throughout the life of this MI function group to request MB operations.

The ctx argument is a pointer to a CMPIContext structure containing the Invocation Context. This context contains the CMPIInitNameSpace property indicating the namespace for which this MI group is to be initialized.

The rc output argument is used to return a CMPIStatus structure containing the service return status.

RETURN VALUE

The return value shall be a pointer to a valid CMPI<mi-type>MI or NULL. NULL indicates that the respective MI group is not supported.

ERRORS

In case rc->rc is not equal to CMPI_RC_OK, the MI is considered malfunctioning and must not be used by the MB.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

_Generic_Create_<mi-type>MI()

NAME

`_Generic_Create_AssociationMI()`

`_Generic_Create_PropertyMI()`

`_Generic_Create_MethodMI()`

`_Generic_Create_IndicationMI()`

SYNOPSIS

```
<mi-type>MI* _Generic_Create_<mi-type>MI(
    const CMPIBroker* broker,
    const CMPIContext* ctx
    const char* miName,
    CMPIStatus* rc);
```

DESCRIPTION

The `_Generic_Create_<mi-type>MI()` functions shall perform any necessary intialization operations of this MI group.

The `broker` argument is a pointer to a `CMPIBroker` structure. This structure can be used throughout the life of this MI function group to request MB operations.

The `ctx` argument is a pointer to a `CMPIContext` structure containing the Invocation Context. This context contains the `CMPIInitNameSpace` property indicating the namespace for which this MI group is to be initialized.

The `miName` argument specifies the name of the MI to be initialized.

The `rc` output argument is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The return value shall be a pointer to a valid `CMPI<mi-type>MI` or NULL. NULL indicates that the respective MI group is not supported.

ERRORS

In case `rc->rc` is not equal to `CMPI_RC_OK`, the MI is considered malfunctioning and must not be used by the MB.

EXAMPLES

None.

APPLICATION USAGE

The `_Generic_Create_<mi-type>MI()` functions are used for MIs that act as single proxy for other MIs. How the proxy mechanism is implemented is up to the MI design. Possible application of proxies could be for remote or out-of-process MIs.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIAssociationMIFT.cleanup()

NAME

`CMPIAssociationMIFT.cleanup()` – perform cleanup prior to unloading the Association provider

SYNOPSIS

```
CMPIStatus CMPIAssociationMIFT.cleanup(
    CMPIAssociationMI* mi,
    const CMPIContext* ctx,
    CMPIBoolean terminating
);
```

DESCRIPTION

The `CMPIAssociationMIFT.cleanup()` function shall perform any necessary cleanup operations prior to the unloading of the library of which this MI group is part. This function is called prior to the unloading of the provider.

The `mi` argument is a pointer to a `CMPIAssociationMI` structure.

The `ctx` argument is a pointer to a `CMPIContext` structure containing the Invocation Context.

When `true`, the `terminating` argument indicates that the MB is in the process of terminating and that cleanup must be done. When set to `false`, the MI may respond with CMPI_RC_DO_NOT_UNLOAD, or CMPI_RC_NEVER_UNLOAD, indicating that unload will interfere with current MI processing.

RETURN VALUE

The `CMPIAssociationMIFT.cleanup()` function returns a `CMPIStatus` structure.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_DO_NOT_UNLOAD	Operation successful – do not unload now.
CMPI_RC_NEVER_UNLOAD	Operation successful – never unload.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIIndicationMIFT.cleanup()

NAME

`CMPIIndicationMIFT.cleanup()` – perform cleanup prior to unloading the Indication provider

SYNOPSIS

```
CMPIStatus CMPIIndicationMIFT.cleanup(
    CMPIIndicationMI* mi,
    const CMPIContext* ctx,
    CMPIBoolean terminating
);
```

DESCRIPTION

The `CMPIIndicationMIFT.cleanup()` function shall perform any necessary cleanup operation prior to the unloading of the library of which this MI group is part. This function is called prior to the unloading of the provider.

The `mi` argument is a pointer to a `CMPIIndicationMI` structure.

The `ctx` argument is a pointer to a `CMPIContext` structure containing the Invocation Context.

When `true`, the `terminating` argument indicates that the MB is in the process of terminating and that cleanup must be done. When set to `false`, the MI may respond with CMPI_RC_DO_NOT_UNLOAD, or CMPI_RC_NEVER_UNLOAD, indicating that unload will interfere with current MI processing.

RETURN VALUE

The `CMPIIndicationMIFT.cleanup()` function returns a `CMPIStatus` structure.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_DO_NOT_UNLOAD	Operation successful – do not unload now.
CMPI_RC_NEVER_UNLOAD	Operation successful – never unload.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIInstanceMIFT.cleanup()

NAME

`CMPIInstanceMIFT.cleanup()` – perform cleanup prior to unloading the Instance provider

SYNOPSIS

```
CMPIStatus CMPIInstanceMIFT.cleanup(
    CMPIInstanceMI* mi,
    const CMPIContext* ctx,
    CMPIBoolean terminating
);
```

DESCRIPTION

The `CMPIInstanceMIFT.cleanup()` function shall perform any necessary cleanup operation prior to the unloading of the library of which this MI group is part. This function is called prior to the unloading of the provider.

The `mi` argument is a pointer to a `CMPIInstanceMI` structure.

The `ctx` argument is a pointer to a `CMPIContext` structure containing the Invocation Context.

When `true`, the `terminating` argument indicates that the MB is in the process of terminating and that cleanup must be done. When set to `false`, the MI may respond with CMPI_RC_DO_NOT_UNLOAD, or CMPI_RC_NEVER_UNLOAD, indicating that unload will interfere with current MI processing.

RETURN VALUE

The `CMPIInstanceMIFT.cleanup()` function returns a `CMPIStatus` structure.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_DO_NOT_UNLOAD	Operation successful – do not unload now.
CMPI_RC_NEVER_UNLOAD	Operation successful – never unload.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIMethodMIFT.cleanup()

NAME

`CMPIMethodMIFT.cleanup()` – perform cleanup prior to unloading the Method provider

SYNOPSIS

```
CMPIStatus CMPIMethodMIFT.cleanup(
    CMPIMethodMI* mi,
    const CMPIContext* ctx,
    CMPIBoolean terminating
);
```

DESCRIPTION

The `CMPIMethodMIFT.cleanup()` function shall perform any necessary cleanup operation prior to the unloading of the library of which this MI group is part. This function is called prior to the unloading of the provider.

The `mi` argument is a pointer to a `CMPIMethodMI` structure.

The `ctx` argument is a pointer to a `CMPIContext` structure containing the Invocation Context.

When `true`, the `terminating` argument indicates that the MB is in the process of terminating and that cleanup must be done. When set to `false`, the MI may respond with CMPI_RC_DO_NOT_UNLOAD, or CMPI_RC_NEVER_UNLOAD, indicating that unload will interfere with current MI processing.

RETURN VALUE

The `CMPIMethodMIFT.cleanup()` function returns a `CMPIStatus` structure.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_DO_NOT_UNLOAD	Operation successful – do not unload now.
CMPI_RC_NEVER_UNLOAD	Operation successful – never unload.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIPropertyMIFT.cleanup()

NAME

`CMPIPropertyMIFT.cleanup()` – perform cleanup prior to unloading the Property provider

SYNOPSIS

```
CMPIStatus CMPIPropertyMIFT.cleanup(
    CMPIMethodMI* mi,
    const CMPIContext* ctx,
    CMPIBoolean terminating
);
```

DESCRIPTION

The `CMPIPropertyMIFT.cleanup()` function is called prior to the unloading of the provider.

The `mi` argument is a pointer to a `CMPIPropertyMI` structure.

The `ctx` argument is a pointer to a `CMPIContext` structure containing the Invocation Context.

When `true`, the `terminating` argument indicates that the MB is in the process of terminating and that cleanup must be done. When set to `false`, the MI may respond with CMPI_RC_DO_NOT_UNLOAD, or CMPI_RC_NEVER_UNLOAD, indicating that unload will interfere with current MI processing.

RETURN VALUE

The `CMPIPropertyMIFT.cleanup()` function returns a `CMPIStatus` structure.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_DO_NOT_UNLOAD	Operation successful – do not unload now.
CMPI_RC_NEVER_UNLOAD	Operation successful – never unload.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

Added in CMPI 2.0.

5.2 Instance MI Signatures

5.2.1 CMPIFlags

The `CMPIFlags` type is used to inform MI functions about options specified by the client and passed on to the MI for certain requests. Normally, MIs will ignore these flags; however, these flags can be useful when MB services are invoked, or an external MB is contacted. `CMPIFlags` are not passed to MIs directly. MIs can use `CMPIContext` services to gain access under the name `CMPIInvocationFlags`.

```
typedef unsigned int CMPIFlags;

#define CMPI_FLAG_LocalOnly 1
#define CMPI_FLAG_DeepInheritance 2
#define CMPI_FLAG_IncludeQualifiers 4
#define CMPI_FLAG_IncludeClassOrigin 8
```

5.2.2 Functions

The Instance MI is accessed through the following functions.

CMPIInstanceMIFT.createInstance()

NAME

`CMPIInstanceMIFT.createInstance()` – create an Instance using an `ObjectPath` as reference

SYNOPSIS

```
CMPIStatus CMPIInstanceMIFT.createInstance(
    CMPIInstanceMI* mi,
    const CMPIContext* ctx,
    const CMPIResult* rslt,
    const CMPIObjectPath* op,
    const CMPIInstance* inst
);
```

DESCRIPTION

The `CMPIInstanceMIFT.createInstance()` function shall create an Instance using an `ObjectPath` as reference.

The `mi` argument is a pointer to a `CMPIInstanceMI` structure.

The `ctx` argument is a pointer to a `CMPIContext` structure containing the Invocation Context.

The `rslt` argument is a pointer to a `CMPIResult` structure that is the result data container.

The `op` argument is a pointer to a `CMPIObjectPath` structure containing namespace, classname, and key components.

The `inst` argument is a pointer to the created Instance.

RETURN VALUE

The `CMPIInstanceMIFT.createInstance()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_ERR_NOT_SUPPORTED	Operation not supported by this MI.
CMPI_RC_ERR_ALREADY_EXISTS	Instance already exists.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

`returnData()`

CHANGE HISTORY

None.

CMPIInstanceMIFT.deleteInstance()

NAME

`CMPIInstanceMIFT.deleteInstance()` – delete an Instance defined by an `ObjectPath`

SYNOPSIS

```
CMPIStatus CMPIInstanceMIFT.deleteInstance(
    CMPIInstanceMI* mi,
    const CMPIContext* ctx,
    const CMPIResult* rslt,
    const CMPIObjectPath* op
);
```

DESCRIPTION

The `CMPIInstanceMIFT.deleteInstance()` function shall create an Instance using an `ObjectPath` as reference.

The `mi` argument is a pointer to a `CMPIInstanceMI` structure.

The `ctx` argument is a pointer to a `CMPIContext` structure containing the Invocation Context.

The `rslt` argument is a pointer to a `CMPIResult` structure that is the result data container.

The `op` argument is a pointer to a `CMPIObjectPath` structure containing namespace, classname, and key components.

RETURN VALUE

The `CMPIInstanceMIFT.deleteInstance()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_ERR_NOT_SUPPORTED	Operation not supported by this MI.
CMPI_RC_ERR_ACCESS_DENIED	Not authorized.
CMPI_RC_ERR_NOT_FOUND	Instance not found.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIInstanceMIFT.enumerateInstanceNames()

NAME

`CMPIInstanceMIFT.enumerateInstanceNames()` – enumerate the `ObjectPaths` of Instances serviced by this provider

SYNOPSIS

```
CMPIStatus CMPIInstanceMIFT.enumerateInstanceNames(
    CMPIInstanceMI* thisMI,
    const CMPIContext* ctx,
    const CMPIResult* rslt,
    const CMPIObjectPath* op
);
```

DESCRIPTION

The `CMPIInstanceMIFT.enumerateInstanceNames()` function shall enumerate the `ObjectPaths` of Instances serviced by this provider.

The `mi` argument is a pointer to a `CMPIInstanceMI` structure.

The `ctx` argument is a pointer to a `CMPIContext` structure containing the Invocation Context.

The `rslt` argument is a pointer to a `CMPIResult` structure that is the result data container.

The `op` argument is a pointer to a `CMPIObjectPath` structure containing namespace and classname components.

RETURN VALUE

The `CMPIInstanceMIFT.enumerateInstanceNames()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes will be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_ERR_NOT_SUPPORTED	Operation not supported by this MI.
CMPI_RC_ERR_ACCESS_DENIED	Not authorized.
CMPI_RC_ERR_NOT_FOUND	Instance not found.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

`returnData()`

CHANGE HISTORY

None.

CMPIInstanceMIFT.enumerateInstances()

NAME

`CMPIInstanceMIFT.enumerateInstances()` – enumerate the Instances serviced by this provider

SYNOPSIS

```
CMPIStatus CMPIInstanceMIFT.enumerateInstances(
    CMPIInstanceMI* thisMI,
    const CMPIContext* ctx,
    const CMPIResult* rslt,
    const CMPIObjectPath* op,
    const char** properties
);
```

DESCRIPTION

The `CMPIInstanceMIFT.enumerateInstances()` function shall enumerate the Instances serviced by this provider.

The `mi` argument is a pointer to a `CMPIInstanceMI` structure.

The `ctx` argument is a pointer to a `CMPIContext` structure containing the Invocation Context.

The `rslt` argument is a pointer to a `CMPIResult` structure that is the result data container.

The `op` argument is a pointer to a `CMPIObjectPath` structure containing namespace and classname components.

The `properties` argument, if not NULL, is an array of elements defining one or more Property names. Each returned Object must not include elements for any Properties missing from this list. The end of the array is identified by a NULL pointer.

RETURN VALUE

The `CMPIInstanceMIFT.enumerateInstances()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_ERR_NOT_SUPPORTED	Operation not supported by this MI.
CMPI_RC_ERR_ACCESS_DENIED	Not authorized.
CMPI_RC_ERR_NOT_FOUND	Instance not found.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

`returnData()`

CHANGE HISTORY

None.

CMPIInstanceMIFT.execQuery()

NAME

`CMPIInstanceMIFT.execQuery()` – query the enumeration of Instances of the class (and subclass) defined by an `ObjectPath`

SYNOPSIS

```
CMPIStatus CMPIInstanceMIFT.execQuery(
    CMPIInstanceMI* mi
    const CMPIContext* ctx,
    const CMPIResult* rslt,
    const CMPIObjectPath* op,
    const char* query,
    const char* lang
);
```

DESCRIPTION

The `CMPIInstanceMIFT.execQuery()` function shall query the enumeration of Instances of the class (and subclass) defined by an `ObjectPath`.

The `mi` argument is a pointer to a `CMPIInstanceMI` structure.

The `ctx` argument is a pointer to a `CMPIContext` structure containing the Invocation Context.

The `rslt` argument is a pointer to a `CMPIResult` structure that is the result data container.

The `op` argument is a pointer to a `CMPIObjectPath` structure containing namespace and classname components.

The `query` argument is a pointer to a string containing the select expression.

The `lang` argument is a pointer to a case-sensitive string containing the query language.

RETURN VALUE

The `CMPIInstanceMIFT.execQuery()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_ERR_NOT_SUPPORTED	Operation not supported by this MI.
CMPI_RC_ERR_ACCESS_DENIED	Not authorized.
CMPI_RC_ERR_QUERY_LANGUAGE_NOT_SUPPORTED	Query language not supported.
CMPI_RC_ERR_INVALID_QUERY	Invalid query.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

`returnData()`

CHANGE HISTORY

None.

CMPIInstanceMIFT.getInstance()

NAME

`CMPIInstanceMIFT.getInstance()` – get the Instance defined by an `ObjectPath`

SYNOPSIS

```
CMPIStatus CMPIInstanceMIFT.getInstance(
    CMPIInstanceMI* thisMI,
    const CMPIContext* ctx,
    const CMPIResult* rslt,
    const CMPIObjectPath* op,
    const char** properties
);
```

DESCRIPTION

The `CMPIInstanceMIFT.getInstance()` function shall get the Instance defined by an `ObjectPath`.

The `mi` argument is a pointer to a `CMPIInstanceMI` structure.

The `ctx` argument is a pointer to a `CMPIContext` structure containing the Invocation Context.

The `rslt` argument is a pointer to a `CMPIResult` structure that is the result data container.

The `op` argument is a pointer to a `CMPIObjectPath` structure containing namespace and classname components.

The `properties` argument, if not NULL, is an array of elements defining one or more Property names. Each returned Object must not include elements for any Properties missing from this list. The end of the array is identified by a NULL pointer.

RETURN VALUE

The `CMPIInstanceMIFT.getInstance()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_ERR_NOT_SUPPORTED	Operation not supported by this MI.
CMPI_RC_ERR_ACCESS_DENIED	Not authorized.
CMPI_RC_ERR_NOT_FOUND	Instance not found.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

`returnData()`

CHANGE HISTORY

None.

CMPIInstanceMIFT.modifyInstance()

NAME

`CMPIInstanceMIFT.modifyInstance()` – replace an existing Instance using an `ObjectPath` as reference

SYNOPSIS

```
CMPIStatus CMPIInstanceMIFT.modifyInstance(
    CMPIInstanceMI* mi,
    const CMPIContext* ctx,
    const CMPIResult* rslt,
    const CMPIObjectPath* op,
    const CMPIInstance* inst,
    const char** properties
);
```

DESCRIPTION

The `CMPIInstanceMIFT.modifyInstance()` function shall replace an existing Instance using an `ObjectPath` as reference.

The `mi` argument is a pointer to a `CMPIInstanceMI` structure.

The `ctx` argument is a pointer to a `CMPIContext` structure containing the Invocation Context.

The `rslt` argument is a pointer to a `CMPIResult` structure that is the result data container.

The `op` argument is a pointer to a `CMPIObjectPath` structure containing namespace, classname, and key components.

The `inst` argument is a pointer to a `CMIPInstance` structure containing the Instance.

The `properties` argument, if not NULL, is an array of elements defining one or more Property names. The process must not replace elements for any Properties missing from this list. If NULL, all properties will be replaced. The end of the array is identified by a NULL pointer.

RETURN VALUE

The `CMPIInstanceMIFT.modifyInstance()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_ERR_NOT_SUPPORTED	Operation not supported by this MI.
CMPI_RC_ERR_ACCESS_DENIED	Not authorized.
CMPI_RC_ERR_NOT_FOUND	Instance not found.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

5.3 Association MI Signatures

The Association MI is accessed using the following functions.

CMPIAssociationMIFT.associatorNames()

NAME

`CMPIAssociationMIFT.associatorNames()` – enumerate `ObjectPaths` of Instances associated with an Instance

SYNOPSIS

```
CMPIStatus CMPIAssociationMIFT.associatorNames(
    CMPIAssociationMI* mi,
    const CMPIContext* ctx,
    const CMPIResult* rslt,
    const CMPIObjectPath* op,
    const char* assocClass,
    const char* resultClass,
    const char* role,
    const char* resultRole
);
```

DESCRIPTION

The `CMPIAssociationMIFT.associatorNames()` function shall enumerate `ObjectPaths` of Instances associated with an Instance.

The `mi` argument points to an Association Instance.

The `ctx` argument points to the Invocation Context.

The `rslt` argument points to the result data container.

The `op` argument points to the source `ObjectPath` containing namespace, classname, and key components.

The `assocClass` argument, if not NULL, shall be a valid Association Class name. It acts as a filter on the returned set of Objects by mandating that each returned Object must be associated to the source Object via an Instance of this Class or one of its subclasses.

The `resultclass` argument, if not NULL, shall be a valid Class name. It acts as a filter on the returned set of Objects by mandating that each returned Object must be either an Instance of this Class (or one of its subclasses).

The `role` argument, if not NULL, shall be a valid Property name. It acts as a filter on the returned set of Objects by mandating that each returned Object must be associated to the source Object via an Association in which the source Object plays the specified role (i.e., the name of the Property in the Association Class that refers to the source Object must match the value of this parameter).

The `resultrole` argument, if not NULL, shall be a valid Property name. It acts as a filter on the returned set of Objects by mandating that each returned Object must be associated to the source Object via an Association in which the returned Object plays the specified role (i.e., the name of the Property in the Association Class that refers to the returned Object must match the value of this parameter).

RETURN VALUE

The `CMPIAssociationMIFT.associatorNames()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_ERR_NOT_SUPPORTED	Operation not supported by this MI.
CMPI_RC_ERR_ACCESS_DENIED	Not authorized.
CMPI_RC_ERR_NOT_FOUND	Instance not found.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIAssociationMIFT.associators()

NAME

`CMPIAssociationMIFT.associators()` – enumerate Instances associated with an Instance

SYNOPSIS

```
CMPIStatus CMPIAssociationMIFT.associators(
    CMPIAssociationMI* mi,
    const CMPIContext* ctx,
    const CMPIResult* rslt,
    const CMPIObjectPath* objName,
    const char* assocClass,
    const char* resultClass,
    const char* role,
    const char* resultRole,
    const char** properties
);
```

DESCRIPTION

The `CMPIAssociationMIFT.associators()` function shall enumerate Instances associated with an Instance.

The `mi` argument points to an Association Instance.

The `ctx` argument points to the Invocation Context.

The `rslt` argument points to the result data container.

The `op` argument points to the source `ObjectPath` containing namespace, classname, and key components.

The `assocClass` argument, if not NULL, shall be a valid Association Class name. It acts as a filter on the returned set of Objects by mandating that each returned Object must be associated to the source Object via an Instance of this Class or one of its subclasses.

The `resultclass` argument, if not NULL, shall be a valid Class name. It acts as a filter on the returned set of Objects by mandating that each returned Object must be either an Instance of this Class (or one of its subclasses).

The `role` argument, if not NULL, shall be a valid Property name. It acts as a filter on the returned set of Objects by mandating that each returned Object must be associated to the source Object via an Association in which the source Object plays the specified role (i.e., the name of the Property in the Association Class that refers to the source Object must match the value of this parameter).

The `resultrole` argument, if not NULL, shall be a valid Property name. It acts as a filter on the returned set of Objects by mandating that each returned Object must be associated to the source Object via an Association in which the returned Object plays the specified role (i.e., the name of the Property in the Association Class that refers to the returned Object must match the value of this parameter).

The `properties` argument, if not NULL, is an array of elements defining one or more Property names. Each returned Object must not include elements for any Properties missing from this list. If NULL, all Properties must be returned. The returned Object must match the value of this parameter. The end of the array is identified by a NULL pointer.

RETURN VALUE

The `CMPIAssociationMIFT.associatorNames()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_ERR_NOT_SUPPORTED	Operation not supported by this MI.
CMPI_RC_ERR_ACCESS_DENIED	Not authorized.
CMPI_RC_ERR_NOT_FOUND	Instance not found.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIAssociationMIFT.referenceNames()

NAME

`CMPIAssociationMIFT.referenceNames()` – enumerate the `ObjectPaths` of Association Instances that refer to an Instance

SYNOPSIS

```
CMPIStatus CMPIAssociationMIFT.referenceNames(
    CMPIAssociationMI* mi,
    const CMPIContext* ctx,
    const CMPIResult* rslt,
    const CMPIObjectPath* op,
    const char* role,
    const char* resultRole
);
```

DESCRIPTION

The `CMPIAssociationMIFT.referenceNames()` function shall enumerate the `ObjectPaths` of Association Instances that refer to an Instance.

The `mi` argument points to an Association Instance.

The `ctx` argument points to the Invocation Context.

The `rslt` argument points to the result data container.

The `op` argument points to the source `ObjectPath` containing namespace, classname, and key components.

The `resultclass` argument, if not NULL, shall be a valid Class name. It acts as a filter on the returned set of Objects by mandating that each returned Object must be either an Instance of this Class (or one of its subclasses).

The `role` argument, if not NULL, shall be a valid Property name. It acts as a filter on the returned set of Objects by mandating that each returned Object must be associated to the source Object via an Association in which the source Object plays the specified role (i.e., the name of the Property in the Association Class that refers to the source Object must match the value of this parameter).

RETURN VALUE

The `CMPIAssociationMIFT.referenceNames()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_ERR_NOT_SUPPORTED	Operation not supported by this MI.
CMPI_RC_ERR_ACCESS_DENIED	Not authorized.
CMPI_RC_ERR_NOT_FOUND	Instance not found.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIAssociationMIFT.references()

NAME

`CMPIAssociationMIFT.references()` – enumerate the Association Instances that refer to an Instance

SYNOPSIS

```
CMPIStatus CMPIAssociationMIFT.references(
    CMPIAssociationMI* mi,
    const CMPIContext* ctx,
    const CMPIResult* rslt,
    const CMPIObjectPath* objName,
    const char* role,
    const char* resultRole,
    const char** properties
);
```

DESCRIPTION

The `CMPIAssociationMIFT.reference()` function shall enumerate the Association Instances that refer to an Instance.

The `mi` argument points to an Association Instance.

The `ctx` argument points to the Invocation Context.

The `rslt` argument points to the result data container.

The `op` argument points to the source `ObjectPath` containing namespace, classname, and key components.

The `resultclass` argument, if not NULL, shall be a valid Class name. It acts as a filter on the returned set of Objects by mandating that each returned Object must be either an Instance of this Class (or one of its subclasses).

The `role` argument, if not NULL, shall be a valid Property name. It acts as a filter on the returned set of Objects by mandating that each returned Object must be associated to the source Object via an Association in which the source Object plays the specified role (i.e., the name of the Property in the Association Class that refers to the source Object must match the value of this parameter).

The `properties` argument, if not NULL, is an array of elements defining one or more Property names. Each returned Object must not include elements for any Properties missing from this list. The end of the array is identified by a NULL pointer.

RETURN VALUE

The `CMPIAssociationMIFT.references()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_ERR_NOT_SUPPORTED	Operation not supported by this MI.
CMPI_RC_ERR_ACCESS_DENIED	Not authorized.

CMPI_RC_ERR_NOT_FOUND	Instance not found.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

5.4 Method MI Signatures

The Method MI is accessed using the following function.

CMPIMethodMIFT.invokeMethod()

NAME

`CMPIMethodMIFT.invokeMethod()` – invoke a named, extrinsic method of an Instance defined by an `ObjectPath`

SYNOPSIS

```
CMPIStatus CMPIMethodMIFT.invokeMethod(
    CMPIMethodMI* mi,
    const CMPIContext* ctx,
    const CMPIResult* rslt,
    const CMPIObjectPath* op,
    const char* method,
    const CMPIArgs* in,
    CMPIArgs* out
);
```

DESCRIPTION

The `CMPIMethodMIFT.invokeMethod()` function shall invoke a named, extrinsic method of an instance defined by a specified `ObjectPath`.

The `mi` argument points to a `CMPIMethodMI` object.

The `ctx` argument points to the Invocation Context object.

The `op` argument points to the source `ObjectPath` containing namespace, classname, and key elements.

The `method` argument points to a string containing the method name.

The `in` argument points to a `CMPIArgs` structure containing the input parameters.

The `out` argument points to a `CMPIArgs` structure containing the output parameters.

RETURN VALUE

The `CMPIMethodMIFT.invokeMethod()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_ERR_NOT_SUPPORTED	Operation not supported by this MI.
CMPI_RC_ERR_ACCESS_DENIED	Not authorized.
CMPI_RC_ERR_NOT_FOUND	Instance not found.
CMPI_RC_ERR_METHOD_NOT_AVAILABLE	Method not available.
CMPI_RC_ERR_METHOD_NOT_FOUND	Method not found.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

`CMPIArgsFT.getArg()`, `CMPIArgsFT.setArg()`, `returnData()`

CHANGE HISTORY

None.

5.5 Indication MI Signatures

The functions defined here are largely modeled after existing CIMOM implementations (Solaris WBEM Services and The Open Group OpenPegasus). They are, currently, the most widely used and therefore a *de facto* standard.

The Indication MI is accessed using the following functions.

CMPIIndicationMIFT.activateFilter()

NAME

`CMPIIndicationMIFT.activateFilter()` – ask the provider to begin monitoring a resource

SYNOPSIS

```
CMPIStatus CMPIIndicationMIFT.activateFilter(
    CMPIIndicationMI* mi,
    const CMPIContext* ctx,
    const CMPISelectExp* filter,
    const char* className,
    const CMPIObjectPath* classPath,
    CMPIBoolean firstActivation
);
```

DESCRIPTION

The `CMPIIndicationMIFT.activateFilter()` function shall begin monitoring the resource according to the filter expression only.

The `mi` argument points to a `CMPIIndicationMI` structure.

The `ctx` argument points to a `CMPIContext` structure containing the Invocation Context.

The `filter` argument contains the filter specification for this subscription to become active.

The `className` argument is the class name extracted from the filter `FROM` clause.

The `classPath` argument is the name of the class for which monitoring is required. Only the namespace part is set if `eventType` is a process indication.

The `firstActivation` argument is set to `true` if this is the first filter for this indication type.

RETURN VALUE

The `CMPIIndicationMIFT.activateFilter()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_ERR_NOT_SUPPORTED	Operation not supported by this MI.
CMPI_RC_ERR_ACCESS_DENIED	Not authorized.
CMPI_RC_ERR_INVALID_QUERY	Invalid query or too complex.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

MIs must not generate indications, unless the MI has been enabled via the `CMPIIndicationMIFT.enableIndications()` function. When disabled via the `CMPIIndicationMIFT.disableIndications()` function, indications must not be generated either.

SEE ALSO

`CMPIBrokerFT.deliverIndication()`

CHANGE HISTORY

None.

CMPIIndicationMIFT.authorizeFilter()

NAME

`CMPIIndicationMIFT.authorizeFilter()` – ask a provider to verify whether this filter is allowed

SYNOPSIS

```
CMPIStatus CMPIIndicationMIFT.authorizeFilter(
    CMPIIndicationMI* mi,
    const CMPIContext* ctx,
    const CMPISelectExp* filter,
    const char* className,
    const CMPIObjectPath* classPath,
    const char* owner
);
```

DESCRIPTION

The `CMPIIndicationMIFT.authorizeFilter()` function shall verify whether this filter is allowed.

The `mi` argument points to a `CMPIIndicationMI` structure.

The `ctx` argument points to a `CMPIContext` structure containing the Invocation Context.

The `filter` argument contains the filter that must be authorized.

The `className` argument is the class name extracted from the filter `FROM` clause.

The `classPath` argument is the name of the class for which monitoring is required. Only the namespace part is set if `eventType` is a process indication.

The `owner` argument is the destination owner.

RETURN VALUE

The `CMPIIndicationMIFT.authorizeFilter()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_ERR_NOT_SUPPORTED	Operation not supported by this MI.
CMPI_RC_ERR_ACCESS_DENIED	Not authorized.
CMPI_RC_ERR_INVALID_QUERY	Invalid query or too complex.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

`returnData()`

CHANGE HISTORY

None.

CMPIIndicationMIFT.deActivateFilter()

NAME

`CMPIIndicationMIFT.deActivateFilter()` – inform the MI that monitoring using this filter should stop

SYNOPSIS

```
CMPIStatus CMPIIndicationMIFT.deActivateFilter(
    CMPIIndicationMI* mi,
    const CMPIContext* ctx,
    const CMPISelectExp* filter,
    const char* className,
    const CMPIObjectPath* classPath,
    CMPIBoolean lastActivation
);
```

DESCRIPTION

The `CMPIIndicationMIFT.deActivateFilter()` function invocation mandates the MI to stop monitoring the resource using this filter.

The `mi` argument points to a `CMPIIndicationMI` structure.

The `ctx` argument points to a `CMPIContext` structure containing the Invocation Context.

The `filter` argument contains the filter that must no longer be used.

The `className` argument is the class name extracted from the filter `FROM` clause.

The `classPath` argument is the name of the class for which monitoring is required. Only the namespace part is set if `eventType` is a process indication.

The `lastActivation` argument is set to `true` if this is the last filter for this indication type.

RETURN VALUE

The `CMPIIndicationMIFT.deActivateFilter()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_ERR_NOT_SUPPORTED	Operation not supported by this MI.
CMPI_RC_ERR_ACCESS_DENIED	Not authorized.
CMPI_RC_ERR_INVALID_QUERY	Invalid query or too complex.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIIndicationMIFT.mustPoll()

NAME

`CMPIIndicationMIFT.mustPoll()` – ask the MI whether polling mode should be used

SYNOPSIS

```
CMPIStatus CMPIIndicationMIFT.mustPoll(
    CMPIIndicationMI* mi,
    const CMPIContext* ctx,
    const CMPIResult* rslt,
    const CMPISelectExp* filter,
    const char* className,
    const CMPIObjectPath* classPath
);
```

DESCRIPTION

The `CMPIIndicationMIFT.mustPoll()` function enables very simple MIs to support indications without providing a complete indication support implementation. When `true` is returned, the MB will enumerate the instances of this MI at regular intervals and apply indication filters.

The `mi` argument points to a `CMPIIndicationMI` structure.

The `ctx` argument points to a `CMPIContext` structure containing the Invocation Context.

The `rslt` argument points to a `CMPIResult` structure used to return a `CMPIBoolean` indicating whether polling mode should be used.

The `filter` argument contains the filter for which this request is made.

The `className` argument is the class name extracted from the filter `FROM` clause.

The `classPath` argument is the name of the class for which monitoring is required. Only the namespace part is set if `eventType` is a process indication.

RETURN VALUE

The `CMPIIndicationMIFT.mustPoll()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_ERR_NOT_SUPPORTED	Operation not supported by this MI.
CMPI_RC_ERR_ACCESS_DENIED	Not authorized.
CMPI_RC_ERR_INVALID_QUERY	Invalid query or too complex.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

`returnData()`

CHANGE HISTORY

None.

CMPIIndicationMIFT.enableIndications()

NAME

`CMPIIndicationMIFT.enableIndications()` – tell the MI that indications can now be generated

SYNOPSIS

```
CMPIStatus CMPIIndicationMIFT.enableIndications(
    CMPIIndicationMI* mi,
    const CMPIContext* ctx,
);
```

DESCRIPTION

The MB is now prepared to process indications.

The `CMPIIndicationMIFT.enableIndications()` function is normally called by the MB after having done its initialization and processing of persistent subscription requests.

The `mi` argument points to a `CMPIIndicationMI` structure.

The `ctx` argument points to a `CMPIContext` structure containing the Invocation Context.

RETURN VALUE

The `CMPIIndicationMIFT.enableIndicationsl()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_ERR_NOT_SUPPORTED	Operation not supported by this MI.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

`returnData()`

CHANGE HISTORY

None.

CMPIIndicationMIFT.disableIndications()

NAME

`CMPIIndicationMIFT.disableIndications()` – tell the MI to stop generating indications

SYNOPSIS

```
CMPIStatus CMPIIndicationMIFT.disableIndications(
    CMPIIndicationMI* mi,
    const CMPIContext* ctx,
);
```

DESCRIPTION

The MB will not accept any indications until enabled again. The `CMPIIndicationMIFT.disableIndications()` function is normally called when the MB is shutting down indication services either temporarily or permanently.

The `mi` argument points to a `CMPIIndicationMI` structure.

The `ctx` argument points to a `CMPIContext` structure containing the Invocation Context.

RETURN VALUE

The `CMPIIndicationMIFT.disableIndications()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_ERR_NOT_SUPPORTED	Operation not supported by this MI.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

`returnData()`

CHANGE HISTORY

None.

5.6 Property MI Signatures

The Property MI is accessed using the following functions.

CMPIPropertyMIFT.setProperty()

NAME

`CMPIPropertyMIFT.setProperty()` – set the named property value of an Instance defined by the `op` argument

SYNOPSIS

```
CMPIStatus CMPIPropertyMIFT.setProperty(
    CMPIPropertyMI* mi,
    const CMPIContext* ctx,
    const CMPIResult* rslt,
    const CMPIObjectPath* op,
    const char* name,
    const CMPIData data)
```

DESCRIPTION

The `CMPIPropertyMIFT.setProperty()` function sets a named property value of an Instance.

The `mi` argument is a pointer to a `CMPIPropertyMI` structure.

The `ctx` argument is a pointer to a `CMPIContext` structure containing the Invocation Context.

The `rslt` argument points to the result data container.

The `op` argument points to the source `ObjectPath` containing namespace, classname, and key components.

The `name` argument points to a string containing the property name.

The `data` argument is a `CMPIData` structure containing the data value to be assigned to the property.

RETURN VALUE

The `CMPIPropertyMIFT.setProperty()` function shall return a `CMPIStatus` structure.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_ERR_ACCESS_DENIED	Not authorized.
CMPI_RC_ERR_NOT_SUPPORTED	Operation not supported by this MI.
CMPI_RC_ERR_INVALID_NAMESPACE	The namespace is invalid.
CMPI_RC_ERR_INVALID_PARAMETER	The parameter is invalid.
CMPI_RC_ERR_INVALID_CLASS	The CIM class does not exist in the specified namespace.
CMPI_RC_ERR_NOT_FOUND	Instance not found.

CMPI_RC_ERR_NO_SUCH_PROPERTY	Entry not found.
CMPI_RC_ERR_TYPE_MISMATCH	Value types incompatible.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

Added in CMPI 2.0.

CMPIPropertyMIFT.getProperty()

NAME

`CMPIPropertyMIFT.getProperty()` – get a named property value of an Instance defined by the `op` argument

SYNOPSIS

```
CMPIStatus CMPIPropertyMIFT.getProperty(
    CMPIPropertyMI* mi,
    const CMPIContext* ctx,
    const CMPIResult* rslt,
    const CMPIObjectPath* op,
    const char* name)
```

DESCRIPTION

The `CMPIPropertyMIFT.getProperty()` function gets a named property value of an Instance.

The `mi` argument is a pointer to a `CMPIPropertyMI` structure.

The `ctx` argument is a pointer to a `CMPIContext` structure containing the Invocation Context.

The `rslt` argument points to the result data container.

The `op` argument points to the source `ObjectPath` containing namespace, classname, and key components.

The `name` argument points to a string containing the property name.

RETURN VALUE

The `CMPIPropertyMIFT.getProperty()` function shall return a `CMPIStatus` structure.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_ERR_ACCESS_DENIED	Not authorized.
CMPI_RC_ERR_INVALID_NAMESPACE	The namespace is invalid.
CMPI_RC_ERR_INVALID_PARAMETER	The parameter is invalid.
CMPI_RC_ERR_INVALID_CLASS	The CIM class does not exist in the specified namespace.
CMPI_RC_ERR_NOT_FOUND	Instance not found.
CMPI_RC_ERR_NO_SUCH_PROPERTY	Entry not found.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

Added in CMPI 2.0.

5.7 CMPI Result Data Support

CMPI enables an MB to actively accept result data as it is generated. All return data produced by MI functions must be returned using the `CMPIResultFT.returnXXX()` functions. Enumerating calls may use series of these calls. Returning data must always be terminated via the `CMPIResultFT.returnDone()` function.

An example for `getProperty()`:

```
CMPIStatus sample_getProperty(
    const CMPIContext* ctx,
    const CMPIResult* rslt,
    const CMPIObjectPath* cop,
    const char* name)
{
    int rv=4711;
    CMReturnData(rslt,&rv,CMPI_sint32);
    CMReturnDone(rslt);
    CMReturn(CMPI_RC_OK);
}
```

An example for `enumerateInstanceNames()` returning a `CMPIObjectPath` enumeration:

```
CMPIBroker* broker;     // Set in <mi-name>_Create_InstanceMI.

CMPIStatus sample_enumerateInstanceNames(
    const CMPIContext* ctx,
    const CMPIResult* rslt,
    const CMPIObjectPath* cop,
    const char** properties)
{
    CMPIObjectPath* p1;
    CMPIString ns;

    for (int i=0; i<3; i++) {
        ns=CMGetNameSpace(cop);
        p1=CMNewObjectPath(broker,CMGetCharPtr(ns),
            "myClass",&rc);
        CMAddKey(p1,"id",&i,CMPI_sint32);
        CMReturnObjectPath(rslt,p1);
    }
    CMReturnDone(rslt);
    CMReturn(CMPI_RC_OK);
}
```

`CMPIResult` support is made available via the `CMPIResultFT` function table:

```
typedef struct _CMPIResult {
    const void* hdl;
    const CMPIResultFT* ft;
} CMPIResult;

typedef struct _CMPIResultFT {
    int ftVersion;
```

```
    CMPIStatus (*release)
        (CMPIResult*);
    CMPIResult* (*clone)
        (const CMPIResult*,CMPIStatus*);
    CMPIStatus (*returnData)
        (const CMPIResult*,const CMPIValue*,const CMPIType);
    CMPIStatus (*returnInstance)
        (const CMPIResult*,const CMPIInstance*);
    CMPIStatus (*returnObjectPath)
        (const CMPIResult*,const CMPIObjectPath*);
    CMPIStatus (*returnDone)
        (const CMPIResult*);
    CMPIStatus (*returnError)
        (const CMPIResult*,const CMPIError*);
} CMPIResultFT;
```

Descriptions of the individual `CMPIResult` functions follow.

CMPIResultFT.clone()

NAME

`CMPIResultFT.clone()` – create an independent copy of a Result object

SYNOPSIS

```
CMPIResult* CMPIResultFT.clone(
    const CMPIResult* rslt,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIResultFT.clone()` function shall create an independent copy of a Result object.

The `rslt` argument points to the `CMPIResult` structure to be copied.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

The resulting Result object must be explicitly released.

RETURN VALUE

The `CMPIResultFT.clone()` function shall return a pointer to the copied Result object. In case of error, NULL shall be returned.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIResultFT.release()

NAME

`CMPIResultFT.release()` – the Result object will not be used any further and may be freed by the CMPI run-time system

SYNOPSIS

```
CMPIStatus CMPIResultFT.release(
    CMPIResult* rslt
);
```

DESCRIPTION

The `CMPIResultFT.release()` function shall indicate that a Result object will not be used any further, and may be freed by the CMPI run-time system.

The `rslt` argument points to a `CMPIResult` structure.

RETURN VALUE

The `CMPIResultFT.release()` function shall return a `CMPIStatus` structure containing the service return status.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIResultFT.returnData()

NAME

`CMPIResultFT.returnData()` – return a value/type pair

SYNOPSIS

```
CMPIStatus CMPIResultFT.returnData(
    const CMPIResult* rslt,
    const CMPIValue* value,
    const CMPIType type
);
```

DESCRIPTION

The `CMPIResultFT.returnData()` function shall accept a value/type pair to be returned to the MB.

The `rslt` argument points to a `CMPIResult` structure containing results to be returned.

The `value` argument points to a `CMPIValue` structure to contain the returned value.

The `type` argument points to a `CMPIType` structure to contain the returned type.

RETURN VALUE

The `CMPIResultFT.returnData()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_DATA_TYPE	An attempt to return a data type that is not allowed for this MI function invocation or type is not recognized.
CMPI_RC_ERR_INVALID_HANDLE	The `rslt` handle is invalid or NULL.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIResultFT.returnDone()

NAME

`CMPIResultFT.returnDone()` – indicate that no further data will be returned by the MI

SYNOPSIS

```
CMPIStatus CMPIResultFT.returnDone(
    const CMPIResult* rslt
);
```

DESCRIPTION

The `CMPIResultFT.returnDone()` function indicates that the MI will not return any more data to the MB.

The `rslt` argument points to a `CMPIResult` structure containing results to be returned.

RETURN VALUE

The `CMPIResultFT.returnDone()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `rslt` handle is invalid or NULL.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

`CMPIResultFT.returnData()`, `invokeMethod()`

CHANGE HISTORY

None.

CMPIResultFT.returnInstance()

NAME

`CMPIResultFT.returnInstance()` – return an Instance object

SYNOPSIS

```
CMPIStatus CMPIResultFT.returnInstance(
    const CMPIResult* rslt,
    const CMPIInstance* inst
);
```

DESCRIPTION

The `CMPIResultFT.returnInstance()` function shall accept an Instance object to be returned to the MB.

The `rslt` argument points to a `CMPIResult` structure containing results to be returned.

The `inst` argument points to a `CMPIInstance` structure to contain the returned value.

RETURN VALUE

The `CMPIResultFT.returnInstance()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `rslt` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIResultFT.returnObjectPath()

NAME

`CMPIResultFT.returnObjectPath()` – return an `ObjectPath` object

SYNOPSIS

```
CMPIStatus CMPIResultFT.returnObjectPath(
    const CMPIResult* rslt,
    const CMPIObjectPath* ref
);
```

DESCRIPTION

The `CMPIResultFT.returnObjectPath()` function shall accept an `ObjectPath` object to be returned to the MB.

The `rslt` argument points to a `CMPIResult` structure containing results to be returned.

The `ref` argument points to a `CMPIObjectPath` structure to contain the returned value.

RETURN VALUE

The `CMPIResultFT.returnObjectPath()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `rslt` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIResultFT.returnError()

NAME

`CMPIResultFT.returnError()`– return a `CMPIError` object

SYNOPSIS

```
CMPIStatus CMPIResultFT.returnError(
    const CMPIResult* rslt,
    const CMPIError* er
);
```

DESCRIPTION

The `CMPIResultFT.returnError()` function shall accept a `CMPIError` object which will be returned to the MB.

The `rslt` argument points to a `CMPIResult` structure containing results to be returned.

The `er` argument points to a `CMPIError` structure to return.

RETURN VALUE

The `CMPIResultFT.returnError()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `rslt` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

5.8 Context Data Support

MBs can pass unstructured data to the MI. The data is organized as name-value pairs. Arrays are supported, but not for type `CMPI_ptr`. Context data support is an optional feature; requests can be terminated by returning CMPI_RC_ERR_NOT_SUPPORTED.

`CMPIContext` can also be used by MBs to carry the MB internal security context. In case MIs use threading and issue CMPI calls from threads, the following functions must be used:

```
CMPIBrokerFT.prepareAttachThread()
CMPIBrokerFT.attachThread()
```

This enables the MB to set up the correct security context for foreign threads.

Currently, the following `CMPIContext` entry names are defined:

`CMPIInitNameSpace`	`CMPI_string`	Namespace for which the MI is started.
`CMPIInvocationFlags`	`CMPI_uint32`	`CMPIFlags` – invocation flags as specified by the client.
`CMPIPrincipal`	`CMPI_string`	Authenticated ID of the user requesting this MI invocation.
`CMPIRole`	`CMPI_string`	The role assumed by the current authenticated user.
`CMPIAcceptLanguage`	`CMPI_string`	A list of language tags (each with an optional quality value), from the corresponding HTTP header of the request, indicating the set of preferred languages in the response. (See RFC 2616 and DSP0200.)
`CMPIContentLanguage`	`CMPI_string`	A list of language tags indicating the language(s) of the content in the HTTP request or response message. (See RFC 2616 and DSP0200.)

`CMPIContext` support is made available via the `CMPIContextFT` function table:

```
typedef struct _CMPIContext {
    const void* hdl;
    const CMPIContextFT* ft;
} CMPIContext;

typedef struct _CMPIContextFT {
    int ftVersion;
    CMPIStatus (*release)
        (CMPIContext*);
    CMPIContext* (*clone)
        (const CMPIContext*,CMPIStatus*);
    CMPIData (*getEntry)
        (const CMPIContext*,const char*,CMPIStatus*);
    CMPIData (*getEntryAt)
        (const CMPIContext*,CMPICount,CMPIString**,
         CMPIStatus*);
    CMPICount (*getEntryCount)
        (const CMPIContext*,CMPIStatus*);
```

```
    CMPIStatus (*addEntry)
        (const CMPIContext*,const char*,const CMPIValue*,
         const CMPIType);
} CMPIContextFT;
```

CMPIContextFT.clone()

NAME

`CMPIContextFT.clone()` – create an independent copy of a Context object

SYNOPSIS

```
CMPIContext* CMPIContextFT.clone(
    const CMPIContext* ctx,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIContextFT.clone()` function shall create an independent copy of a Context object.

The `ctx` argument points to the `CMPIContext` structure to be copied.

The `rc` argument points to a `CMPIStatus` structure containing the service return status.

The resulting Context object must be explicitly released.

RETURN VALUE

The `CMPIContextFT.clone()` function shall return a pointer to the copied Context object. In case of error, NULL shall be returned.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `ctx` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIContextFT.release()

NAME

`CMPIContextFT.release()` – the Context object will not be used any further and may be freed by the CMPI run-time system

SYNOPSIS

```
CMPIStatus CMPIContextFT.release(
    CMPIContext* ctx
);
```

DESCRIPTION

The `CMPIContextFT.release()` function shall indicate that a Context object will not be used any further, and may be freed by the CMPI run-time system.

The `ctx` argument points to a `CMPIContext` structure.

RETURN VALUE

The `CMPIContextFT.release()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `ctx` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIContextFT.addEntry()

NAME

`CMPIContextFT.addEntry()` – add/replace a named Context entry

SYNOPSIS

```
CMPIStatus CMPIContextFT.addEntry(
    const CMPIContext* ctx,
    const char* name;
    const CMPIValue* data,
    const CMPIType type
);
```

DESCRIPTION

The `CMPIContextFT.addEntry()` function shall add or replace a named Context entry.

The `ctx` argument points to a `CMPIContext` structure.

The `name` argument is a string containing the context entry name.

The `data` argument points to a `CMPIData` structure containing the data to be assigned to the context entry.

The `type` argument is a `CMPIType` that defines the type of the data.

RETURN VALUE

The `CMPIContextrFT.addEntry()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_DATA_TYPE	Type not supported for this call, or type is not recognized.
CMPI_RC_ERR_INVALID_HANDLE	The `ctx` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIContextFT.getEntry()

NAME

`CMPIContextFT.getEntry()` – get a Context entry by name

SYNOPSIS

```
CMPIData CMPIContextFT.getEntry(
    const CMPIContext* ctx,
    const char* name,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIContextFT.getEntry()` function shall get a named Context entry.

The `ctx` argument points to a `CMPIContext` structure.

The `name` argument is a string containing the context entry name.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIContextFT.getEntry()` function shall return a `CMPIData` structure containing the Context entry value.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_NO_SUCH_PROPERTY	Entry not found.
CMPI_RC_ERR_INVALID_HANDLE	The `ctx` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIContextFT.getEntryAt()

NAME

`CMPIContextFT.getEntryAt()` – get a Context entry defined by its index

SYNOPSIS

```
CMPIData CMPIContextFT.getEntryAt(
    const CMPIContext* ctx,
    CMPICount index,
    CMPIString** name,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIContextFT.getEntryAt()` function shall get a Context defined by its index.

The `ctx` argument points to a `CMPIContext` structure.

The `index` argument specifies the zero-based position of the context entry in the internal data array.

The `name` output argument, if not NULL, is used to return a pointer to a `CMPIString` structure containing the name of the returned Context entry.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIContextFT.getEntryAt()` function shall return a `CMPIData` structure containing the Context entry value.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_NO_SUCH_PROPERTY	The index value is out of range.
CMPI_RC_ERR_INVALID_HANDLE	The `ctx` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIContextFT.getEntryCount()

NAME

`CMPIContextFT.getEntryCount()` – get the number of entries contained in a context

SYNOPSIS

```
CMPICount CMPIContextFT.getEntryCount(
    const CMPIContext* ctx,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIContextFT.getEntryCount()` function shall get the number of entries contained in a context.

The `ctx` argument points to a `CMPIContext` structure.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIContextFT.getEntryCount()` function shall return a count of the number of entries in the Context. In case of error, the return value shall be undefined.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `ctx` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

6 Data Type Manipulation Functions

Data type manipulation support is used to encapsulate Management Broker (MB)-specific implementation details. Support is provided for the following data types:

```
CMPIInstance
CMPIObjectPath
CMPIArgs
CMPIEnumeration
CMPISelectExp
CMPISelectCond
CMPISubCond
CMPIPredicate
```

6.1 CMPIBrokerEnc Support

Most of the functions available via `CMPIBrokerEnc` are factory services for the CMPI data types. Other functions deal with debugging and logging support. The `CMPIBrokerEncFT` is the function table used as an anchor for these services. It is made available to Management Instrumentation (MI) via the `<mi-name>_Create_<mi-type>Mi()` functions.

```
typedef struct _CMPIBrokerEncFT {
    int ftVersion;
    CMPIInstance* (*newInstance)
        (const CMPIBroker*,
         const CMPIObjectPath*,CMPIStatus*);
    CMPIObjectPath* (*newObjectPath)
        (const CMPIBroker*,
         const char*,const char*,CMPIStatus*);
    CMPIArgs* (*newArgs)
        (const CMPIBroker*,CMPIStatus*);
    CMPIString* (*newString)
        (const CMPIBroker*,const char*,CMPIStatus*);
    CMPIArray* (*newArray)
        (const CMPIBroker*,CMPICount,CMPIType,CMPIStatus*);
    CMPIDateTime* (*newDateTime)
        (const CMPIBroker*,CMPIStatus*);
    CMPIDateTime* (*newDateTimeFromBinary)
        (const CMPIBroker*,CMPIUint64,CMPIBoolean,
         CMPIStatus*);
    CMPIDateTime* (*newDateTimeFromChars)
        (const CMPIBroker*,const char*,CMPIStatus*);
    CMPISelectExp* (*newSelectExp)
        (const CMPIBroker*,const char*,const char*,
         CMPIArray**,CMPIStatus*);
    CMPIBoolean (*classPathIsA)
        (const CMPIBroker*,const CMPIObjectPath*,
         const char*,CMPIStatus*);
```

```
    CMPIString* (*toString)
        (const CMPIBroker*,const void*,CMPIStatus*);
    CMPIBoolean (*isOfType)
        (const CMPIBroker*,const void*,
         const char*,CMPIStatus*);
    CMPIString* (*getType)
        (const CMPIBroker*,const void*,CMPIStatus*);
    CMPIString* (*getMessage)
        (const CMPIBroker*,const char*,
         const char*,CMPIStatus*,
         CMPICount, …);
    CMPIStatus (*logMessage)
        (const CMPIBroker*,int,const char*,const char*,
         const CMPIString*);
    CMPIStatus  (*trace)
        (const CMPIBroker*,int,const char*,const char*,
         const CMPIString*);
    CMPIError* (*newCMPIError)(const CMPIBroker*,
         const char*,  const char*,  const
         char*, const CMPIErrorSeverity,
         const CMPIErrorProbableCause, const CMPIrc,
             CMPIStatus*);
    CMPIStatus (*openMessageFile)(const CMPIBroker*,
         const char*, CMPIMsgFileHandle*);
    CMPIStatus (*closeMessageFile)(const CMPIBroker*,
         const CMPIMsgFileHandle);
    CMPIString* (*getMessage2)(const CMPIBroker*,
         const char*,
         const CMPIMsgFileHandle, const char*, CMPIStatus*,
         CMPICount, ...);
} CMPIBrokerEncFT;
```

Descriptions of the individual `CMPIBrokerEnc` functions follow.

CMPIBrokerEncFT.newInstance()

NAME

`CMPIBrokerEncFT.newInstance()` – create a new Instance object

SYNOPSIS

```
CMPIInstance* CMPIBrokerEncFT.newInstance(
    const CMPIBroker* mb,
    const CMPIObjectPath* op,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIBrokerEncFT.newInstance()` function shall return a new `CMPIInstance` object.

The `mb` argument points to a `CMPIBroker` object.

The `op` argument points to a `CMPIObjectPath` structure containing the namespace and classname.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIBrokerEncFT.newInstance()` function shall return a pointer to a new `CMPIInstance` object. In case an error is detected, NULL must be returned.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_NAMESPACE	The namespace implied by `op` is invalid.
CMPI_RC_ERR_NOT_FOUND	The class implied by `op` is not found.
CMPI_RC_ERR_INVALID_HANDLE	Either the `mb` or `op` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerEncFT.newObjectPath()

NAME

`CMPIBrokerEncFT.newObjectPath()` – create a new `ObjectPath` object

SYNOPSIS

```
CMPIObjectPath* CMPIBrokerEncFT.newObjectPath(
    const CMPIBroker* mb,
    const char* ns,
    const char* cn,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIBrokerEncFT.newObjectPath()` function shall return a new `CMPIObjectPath` object.

The `mb` argument points to a `CMPIBroker` object.

The `ns` argument points to a string containing the namespace.

The `cn` argument points to a string containing the classname.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIBrokerEncFT.newObjectPath()` function shall return a pointer to a new `CMPIObjectPath` object. NULL shall be returned in case an error is detected.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_NAMESPACE	The namespace `ns` is invalid.
CMPI_RC_ERR_NOT_FOUND	The class `cn` is not found.
CMPI_RC_ERR_INVALID_HANDLE	The `mb` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerEncFT.newArgs()

NAME

`CMPIBrokerEncFT.newArgs()` – create a new `Args` object

SYNOPSIS

```
CMPIArgs* CMPIBrokerEncFT.newArgs(
    const CMPIBroker* mb,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIBrokerEncFT.newArgs()` function shall return a new `CMPIArgs` object.

The `mb` argument points to a `CMPIBroker` object.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIBrokerEncFT.newArgs()` function shall return a pointer to a new `CMPIArgs` object. NULL shall be returned in case an error is detected.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `mb` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerEncFT.newString()

NAME

`CMPIBrokerEncFT.newString()` – create a new String object

SYNOPSIS

```
CMPIString* CMPIBrokerEncFT.newString(
    const CMPIBroker* mb,
    const char* data,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIBrokerEncFT.newString()` function shall return a new `CMPIString` object.

The `mb` argument points to a `CMPIBroker` object.

The `data` argument is a pointer to the data to initialize the new `CMPIString` structure.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIBrokerEncFT.newString()` function shall return a pointer to a new `CMPIString` object. NULL shall be returned in case an error is detected.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `mb` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerEncFT.newArray()

NAME

`CMPIBrokerEncFT.newArray()` – create a new Array object

SYNOPSIS

```
CMPIArray* CMPIBrokerEncFT.newArray(
    const CMPIBroker* mb,
    CMPICount max,
    CMPIType type,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIBrokerEncFT.newArray()` function shall return a new `CMPIArray` object. Once created, the size of the array is fixed and all elements are of the same type.

The `mb` argument points to a `CMPIBroker` object.

The `max` argument defines the maximum number of elements.

The `type` argument specifies the element type.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIBrokerEncFT.newArray()` function shall return a pointer to a new `CMPIArray` object. NULL shall be returned in case an error is detected.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `mb` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerEncFT.newDateTime()

NAME

`CMPIBrokerEncFT.newDateTime()` – create a new `DateTime` object initialized to the current date and time

SYNOPSIS

```
CMPIDateTime* CMPIBrokerEncFT.newDateTime(
    const CMPIBroker* mb,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIBrokerEncFT.newDateTime()` function shall return a new `CMPIDateTime` object initialized with the current date and time.

The `mb` argument points to a `CMPIBroker` object.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIBrokerEncFT.newDateTime()` function shall return a pointer to a new `CMPIDateTime` object. NULL shall be returned in case an error is detected.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `mb` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerEncFT.newDateTimeFromBinary()

NAME

`CMPIBrokerEncFT.newDateTimeFromBinary()` – create a new `DateTime` object initialized to a specified value

SYNOPSIS

```
CMPIDateTime* CMPIBrokerEncFT.newDateTimeFromBinary(
    const CMPIBroker* mb,
    CMPIUint64 binTime,
    CMPIBoolean interval,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIBrokerEncFT.newDateTimeFromBinary()` function shall return a new `CMPIDateTime` object initialized with the specified date and time.

The `mb` argument points to a `CMPIBroker` object.

When `interval` is `false`, the `binTime` argument contains a date/time value expressed as a 64-bit unsigned integer in microseconds starting since 00:00:00 GMT, Jan 1, 1970. Otherwise, the `binTime` argument contains a time interval as a 64-bit unsigned integer in microseconds.

The `interval` argument, if true, indicates that the `binTime` argument is considered to be a time interval in microseconds.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIBrokerEncFT.newDateTimeFromBinary()` function shall return a pointer to a new `CMPIDateTime` object. NULL shall be returned in case an error is detected.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `mb` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerEncFT.newDateTimeFromChars()

NAME

`CMPIBrokerEncFT.newDateTimeFromChars()` – create a new `DateTime` object initialized to a specified value

SYNOPSIS

```
CMPIDateTime* CMPIBrokerEncFT.newDateTimeFromChars(
    const CMPIBroker* mb,
    const char* utcTime,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIBrokerEncFT.newDateTimeFromChars()` function shall return a new `CMPIDateTime` object initialized with the specified date and time. The `utcTime` argument contains a date/time value expressed in the character format defined in the CIM XML v2 specification.

The `utcTime` argument contains a date/time value expressed in UTC character format: `yyymmddhhmmss.mmmmmmsutc` (year, month, day, hour, minute, second, microseconds, and signed UTC offset). Interval times are indicated by `:000` for the signed UTC offset.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIBrokerEncFT.newDateTimeFromChars()` function shall return a pointer to a new `CMPIDateTime` object. NULL shall be returned in case an error is detected.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_PARAMETER	The `utcTime` format is invalid.
CMPI_RC_ERR_INVALID_HANDLE	The `mb` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerEncFT.newSelectExp()

NAME

`CMPIBrokerEncFT.newSelectExp()` – create a new `SelectExp` object

SYNOPSIS

```
CMPISelectExp* CMPIBrokerEncFT.newSelectExp(
    const CMPIBroker* mb,
    const char* query,
    const char* lang,
    const CMPIArray** projection,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIBrokerEncFT.newSelectExp()` function shall return a new `CMPISelectExp` object.

The `mb` argument points to a `CMPIBroker` object.

The `query` argument is a pointer to a string containing the select expression.

The `lang` argument is a pointer to a string containing the query language.

The `projection` output argument is a pointer to a `CMPIArray` structure of `CMPIString` entries containing the projection specification. It shall be set to NULL if no projection is defined. The projection specification is query language-specific. Hence the entries format of the projection output array `CMPIString` might be different depending on the query language. Be sure to check the `lang` argument for the query language your provider will support.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIBrokerEncFT.newSelectExp()` function shall return a pointer to a new `CMPISelectExp` object. NULL shall be returned in case an error is detected.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_QUERY_LANGUAGE_NOT_SUPPORTED	`lang` is not supported.
CMPI_RC_ERR_INVALID_QUERY	The query expression is not valid.
CMPI_RC_ERR_INVALID_HANDLE	The `mb` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerEncFT.classPathIsA()

NAME

`CMPIBrokerEncFT.classpathIsA()` – determine whether a CIM class is of a specified type or any of its subclasses

SYNOPSIS

```
CMPIBoolean CMPIBrokerEncFT.classPathIsA(
    const CMPIBroker* mb,
    const CMPIObjectPath* op,
    const char* type,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIBrokerEncFT.classpathIsA()` function shall determine whether a CIM class is of a specified type or any of that type's subclasses.

The `mb` argument points to a `CMPIBroker` structure.

The `op` argument points to a `CMPIObjectPath` structure.

The `type` argument points to the type to be tested for.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIBrokerEncFT.classPathIsA()` function shall return true if the test is successful; `false` shall be returned in case an error is detected.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_NAMESPACE	The namespace implied by `op` is invalid.
CMPI_RC_ERR_NOT_FOUND	The class implied by `op` is not found.
CMPI_RC_ERR_INVALID_PARAMETER	The `type` format is invalid.
CMPI_RC_ERR_INVALID_HANDLE	Either the `mb` or `op` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerEncFT.toString()

NAME

`CMPIBrokerEncFT.toString()` – attempt to transform a CMPI object into an implementation-specific string representation

SYNOPSIS

```
CMPIString* CMPIBrokerEncFT.toString(
    const CMPIBroker* broker,
    const void* object,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIBrokerEncFT.toString()` function attempts to transform a string into an MB implementation-specific string representation.

The `mb` argument is a pointer to a `CMPIBroker` structure.

The `object` argument is a pointer to a valid CMPI object.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIBrokerEncFT.toString()` function returns a pointer to a `CMPIString` structure containing the string representation. Output will vary depending on the specific implementation. NULL shall be returned in case an error is detected.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_PARAMETER	The `type` format is invalid.
CMPI_RC_ERR_INVALID_HANDLE	Either the `mb` or `obj` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

This function is intended for debugging purposes only.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerEncFT.isOfType()

NAME

`CMPIBrokerEncFT.isOfType()` – verify whether an object is of a specified CMPI type

SYNOPSIS

```
CMPIBoolean CMPIBrokerEncFT.isOfType(
    const CMPIBroker* mb,
    const void* object,
    const char* type,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIBrokerEncFT.isOfType()` function shall verify whether an object is of a specified CMPI type.

The `mb` argument points to a `CMPIBroker` structure.

The `object` argument is a pointer to a valid CMPI object.

The `type` argument points to a string specifying a valid CMPI object type to be tested for.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIBrokerEncFT.isOfType()` function shall return true if the test is successful; `false` shall be returned in case an error is detected.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_PARAMETER	The `type` format is invalid.
CMPI_RC_ERR_INVALID_HANDLE	Either the `mb` or `obj` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerEncFT.getType()

NAME

`CMPIBrokerEncFT.getType()` – retrieve the CMPI type of an object

SYNOPSIS

```
CMPIString* CMPIBrokerEncFT.getType(
    const CMPIBroker* mb,
    const void* obj,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIBrokerEncFT.getType()` function shall return the CMPI type of an object.

The `mb` argument points to a `CMPIBroker` structure.

The `obj` argument is a pointer to a valid CMPI object.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIBrokerEncFT.getType()` function shall return a pointer to a `CMPIString` structure containing the encapsulated object type. NULL shall be returned in case an error is detected.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	Either the `mb` or `obj` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerEncFT.getMessage()

NAME

`CMPIBrokerEncFT.getMessage()` – retrieve a translated message depending on the MB language setting

SYNOPSIS

```
CMPIString* CMPIBrokerEncFT.getMessage(
    const CMPIBroker* mb,
    const char* msgId,
    const char* defMsg,
    CMPIStatus* rc,
    CMPICount count,
    ...
);
```

DESCRIPTION

The `CMPIBrokerEncFT.getMessage()` function shall either retrieve a translated message depending on the MB language setting, or return a default message when translation services are not supported, or a message template is not found.

The `mb` argument points to a `CMPIBroker` structure.

The `msgId` argument is used by the message translation service to locate a message template.

The `defMsg` argument, the default message used when the message translation service is not supported, or `msgId` cannot be located. Up to ten message insert triggers ($0 through $9) can be interspersed in the text.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

The `count` argument defines the number of message insert pairs in the range 0 to 10.

Count is followed by 0 to 10 message insert pairs. Each pair has the following format: `CMPIType,value`, whereby `value` must correspond to `CMPIType`. The following value types are supported: `CMPI_sint32`, `CMPI_uint32`, `CMPI_sint64`, `CMPI_uint64`, `CMPI_real64`, `CMPI_Boolean`, `CMPI_chars`, and `CMPI_string`.

RETURN VALUE

The `CMPIBrokerEncFT.getMessage()` function returns a pointer to a `CMPIString` structure containing either the translated or default message. The default message without insert resolution shall be returned in case an error is detected.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_TYPE_MISMATCH	Invalid insert pair.
CMPI_RC_ERR_INVALID_PARAMETER	`count` value range violation.
CMPI_RC_ERR_INVALID_HANDLE	The `mb` handle is invalid.

ERRORS

None.

EXAMPLES

As an example, consider the following:

```
CMGetMessage(_broker,"msg79",
    “Good morning $0 $1, event $2 just occurred.”,&rc,
     3,CMPI_chars,”Peter”,
     CMPI_chars,”Schwarz”,
     CMPI_sint32,79);
```

When no translation services are supported, the following string will be returned:

```
“Good morning Peter Schwarz, event 79 just occurred.”
```

When translation services are supported, then depending on the MB language setting and the message templates, the following messages might be returned:

```
“Morning, just caught event 79.”
“Guten
Morgen Peter, Ereignis 79 ist soeben aufgetreten.
”
```

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerEncFT.logMessage()

NAME

`CMPIBrokerEncFT.logMessage()` – instruct the MB to log a message

SYNOPSIS

```
CMPIStatus CMPIBrokerEncFT.logMessage(
    const CMPIBroker* mb,
    int severity,
    const char* id,
    const char* text,
    const CMPIString* string
);
```

DESCRIPTION

The `CMPIBrokerEncFT.logMessage()` function shall instruct the MB to log a diagnostic message. This function exists to provide a mechanism by which to provide information about errors.

The `mb` argument points to a `CMPIBroker` structure.

The `severity` argument describes the severity of log message. Severity levels are defined in Section 4.8.

The `id` argument, if not NULL, can be a message ID or any other identifying string.

The `text` argument, if not NULL, is the message text to be logged.

The `string` argument, if not NULL, is the message text to be logged. `string` will be ignored when text is not NULL.

RETURN VALUE

The `CMPIBrokerEncFT.logMessage()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_NOT_SUPPORTED	Logging is not supported.
CMPI_RC_ERR_INVALID_HANDLE	Either the `mb` or string handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerEncFT.trace()

NAME

`CMPIBrokerEncFT.trace()` – instruct the MB to accept a trace entry

SYNOPSIS

```
CMPIStatus CMPIBrokerEncFT.trace(
    const CMPIBroker* mb,
    int level,
    const char* component,
    const char* text,
    const CMPIString* string
);
```

DESCRIPTION

The `CMPIBrokerEncFT.trace()` function shall instruct the MB to accept a trace entry. This function exists to provide a mechanism by which debugging information may be generated.

The `mb` argument points to a `CMPIBroker` structure.

The `level` argument describes the level of log message. Levels are defined in Section 4.9.

The `component` argument, if not NULL, is the component ID.

The `text` argument, if not NULL, is the message text to be logged.

The `string` argument, if not NULL, is the message text to be logged. `string` will be ignored when text is not NULL.

RETURN VALUE

The `CMPIBrokerEncFT.trace()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_NOT_SUPPORTED	Tracing is not supported.
CMPI_RC_ERR_INVALID_HANDLE	Either the `mb` or string handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerEncFT.newCMPIError()

NAME

`CMPIBrokerEncFT.newCMPIError()` – instruct CMPI to create a new `CMPIError` object

SYNOPSIS

```
CMPIError* CMPIBrokerEncFT.newCMPIError(
    const CMPIBroker* mb,
    const char* owner,
    const char* msgID,
    const char* msg,
    const CMPIErrorSeverity sev,
    const CMPIErrorProbableCause pc,
    const CMPIrc cimStatusCode,
    CMPIStatus* rc
);
```

DESCRIPTION

This function shall instruct the MB to create a new `CMPIError` object.

The `mb` argument points to a `CMPIBroker` structure.

The `owner` argument points to a string which uniquely identifies the entity that owns the definition of the format of the message. It *must* include a copyrighted, trademarked, or otherwise unique name that is not owned by the business entity or standard body defining the format.

The `msgID` argument points to a string which uniquely identifies, within the scope of the `owner` argument, the format of the message. This is an opaque string supplied by the MI and is not directly used by the MB.

The `msg` argument points to a string which is the formatted and translated message.

The `sev` argument indicates the perceived severity of the error.

The `pc` argument indicates the probable cause of this error.

The `CIMtatusCode` argument identifies the status code which characterizes this instance. Note that not all status codes are valid for each operation. The specification for the operation *should* define the status codes that can be returned.

The `rc` output argument, if not NULL, is used to return a `CMPIStatusstructure` containing the service return status.

RETURN VALUE

The function will return a pointer to the newly allocated `CMPIError` object. In case of error, NULL will be returned. To see if this function failed you *must* inspect the `rc` argument.

CMPI_RC_OK	Indicates the `CMPIError` object was successfully created.
CMPI_RC_ERR_INVALID_HANDLE	The `mb` argument is invalid.
CMPI_RC_ERR_INVALID_PARAMETER	One of the parameters is invalid.

CMPI_RC_ERR_NOT_SUPPORTED	`CMPIError` is not supported.
CMPI_RC_ERR_FAILED	A generic error occurred.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

`getMessage2`

CHANGE HISTORY

None.

CMPIBrokerEncFT.openMessageFile()

NAME

`CMPIBrokerEncFT.openMessageFile()` – open a message file and return a handle to it

SYNOPSIS

```
CMPIStatus CMPIBrokerEncFT.openMessageFile(
    const CMPIBroker* mb,
    const char* msgFile,
    CMPIMsgFileHandle* msgFileHandle
);
```

DESCRIPTION

The `CMPIBrokerEncFT.openMessageFile()` function shall open a message file supported by translation services and return a unique identifier to that file. This function will use the `CMPIAcceptLanguage` entry from the current context to determine the message file to open, and also add the `CMPIContentLanguage` entry to the current context based on the actual message file that was opened. All subsequent calls to the `CMPIBrokerEncFT.getMessage2()` function using this `msgFileHandle` will be associated with this `ContentLanguage`.

The `mb` argument points to a `CMPIBroker` structure.

The `msgFile` argument contains the implementation-specific file path to the message file.

The `msgFileHandle` output argument contains a unique identifier to the open message file that can be passed to the `CMPIBrokerEncFT.getMessage2()` and `CMPIBrokerEncFT.closeMessageFile()` functions.

RETURN VALUE

The `CMPIBrokerEncFT.openMessageFile()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_NOT_FOUND	The message file was not found.
CMPI_RC_ERR_INVALID_HANDLE	The `mb` handle is invalid.

Note that for any error conditions returned by `CMPIBrokerEncFT.openMessageFile()`, a subsequent call to `CMPIBrokerEncFT.getMessage2()` using the `CMPIMsgFileHandle` that was returned will result in the default message being used.

ERRORS

None.

EXAMPLES

```
CMPIStatus rc;
CMPIMsgFileHandle msgFileHandle;
rc = _broker->eft->openMessageFile(_broker,
    "/mymsgs/mybundle", &msgFileHandle);
```

For this example, where the implementation is using ICU translation services, the path to the ICU message bundle is specified as a file system pathname without the language or file extensions (e.g., `/mymsgs/mybundle_fr.res`.)

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerEncFT.closeMessageFile()

NAME

`CMPIBrokerEncFT.closeMessageFile()` – close a message file

SYNOPSIS

```
CMPIStatus CMPIBrokerEncFT.closeMessageFile(
    const CMPIBroker* mb,
    const CMPIMsgFileHandle msgFileHandle
);
```

DESCRIPTION

The `CMPIBrokerEncFT.closeMessageFile()` function shall close a message file previously opened by `CMPIBrokerEncFT.openMessageFile()`.

The `mb` argument points to a `CMPIBroker` structure.

The `msgFileHandle` argument contains a unique identifier to the open message file that was returned by a previous call to `CMPIBrokerEncFT.openMessageFile()`.

RETURN VALUE

The `CMPIBrokerEncFT.closeMessageFile()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	Either the `mb` or `msgFileHandle` handle is invalid.

ERRORS

None.

EXAMPLES

```
CMPIStatus rc;
CMPIMsgFileHandle msgFileHandle;

rc = _broker->eft->openMessageFile(_broker,
    "/mymsgs/mybundle", &msgFileHandle);
rc = _broker->eft->closeMessageFile(_broker, msgFileHandle);
```

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerEncFT.getMessage2()

NAME

`CMPIBrokerEncFT.getMessage2()` – retrieve a translated message

SYNOPSIS

```
CMPIString* CMPIBrokerEncFT.getMessage2(
    const CMPIBroker* mb,
    const char* msgId,
    const CMPIMsgFileHandle msgFileHandle,
    const char* defMsg,
    CMPIStatus* rc,
    CMPICount count,
    ...
);
```

DESCRIPTION

The `CMPIBrokerEncFT.getMessage2()` function shall either retrieve a translated message from the open message file associated with the specified identifier, or return a default message if the message is not found.

The `mb` argument points to a `CMPIBroker` structure.

The `msgId` argument is used by the message translation service to locate a message template.

The `msgFileHandle` argument contains a unique identifier to the open message file that was returned by a previous call to `CMPIBrokerEncFT.openMessageFile()`.

The `defMsg` argument is the default message used when the message translation service is not supported or msgId cannot be located. Up to ten message insert triggers ($0 through $9) can be interspersed in the text.

The `rc` output argument, if not NULL, is used to return a `CMPIStatusstructure` containing the service return status.

The `count` argument defines the number of message insert pairs. Each pair has the following format:

`CMPIType,value`

whereby `value` must correspond to `CMPIType`. The following value types are supported: `CMPI_Sint32`, `CMPI_uint32`, `CMPI_sint64`, `CMPI_uint64`, `CMPI_real64`, `CMPI_Boolean`, `CMPI_chars`, and `CMPI_string`.

RETURN VALUE

The `CMPIBrokerEncFT.getMessage2()` function shall return a pointer to a `CMPIString` structure containing either the translated or default message. The default message without insert resolution shall be returned in case an error is detected.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_TYPE_MISMATCH	Invalid insert pair.

CMPI_RC_INVALID_PARAMETER	Count value range violation.
CMPI_RC_ERR_INVALID_HANDLE	Either the `mb` or `msgFileHandle` handle is invalid.

ERRORS

None.

EXAMPLES

```
CMPIStatus rc;
CMPIString* myMsg;
CMPIMsgFileHandle msgFileHandle;
rc = _broker->eft->openMessageFile(_broker,
    "/mymsgs/mybundle", &msgFileHandle);
myMsg = _broker->eft->getMessage2(_broker, "msg79",
    msgFilehandle,
    "Good morning $0 $1, event $2 just occurred.",
    &rc,
    3,
    CMPI_chars,"Peter",
    CMPI_chars,"Schwarz",
    CMPI_sint32,79);
```

When no translation services are supported, or the message cannot be found, the following string will be returned:

```
"Good morning Peter Schwarz, event 79 just occurred."
```

When translation services are supported and the message is found, then these are a couple examples of messages that might be returned:

```
"Morning, just caught event 79."
"Guten Morgen Peter, Ereignis 79 ist soeben aufgetreten."
```

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

6.2 CMPIString Support

CMPIString support is made available via the CMPIStringFT function table:

```
typedef struct _CMPIString {
    const void* hdl;
    const CMPIStringFT* ft;
} CMPIString;

typedef struct _CMPIStringFT {
    int ftVersion;
    CMPIStatus (*release)
       (CMPIString*);
    CMPIString* (*clone)
       (const CMPIString*,CMPIStatus*);
    const char* (*getCharPtr)
       (const CMPIString*,CMPIStatus*);
} CMPIStringFT;
```

The factory function for new CMPIStrings is CMPIBrokerEncFT.newString().

A detailed description of each CMPIString function follows.

CMPIStringFT.clone()

NAME

`CMPIStringFT.clone()` – create an independent copy of a String object

SYNOPSIS

```
CMPIString* CMPIStringFT.clone(
    const CMPIString* str,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIStringFT.clone()` function shall create an independent copy of a String object.

The `str` argument points to the `CMPIString` structure to be copied.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIStringFT.clone()` function shall return a pointer to the copied String object. NULL shall be returned in case an error is detected.

The resulting `CMPIString` object must be explicitly released using `CMPIStringFT.release()`.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `str` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIStringFT.release()

NAME

`CMPIStringFT.release()` – indicate that the String object will not be used any further and may be freed by the CMPI run-time system

SYNOPSIS

```
CMPIStatus CMPIStringFT.release(
    CMPIString* str
);
```

DESCRIPTION

The `CMPIStringFT.release()` function shall indicate that a String object will not be used any further, and may be freed by the CMPI run-time system.

The `str` argument points to a `CMPIString` structure.

RETURN VALUE

The `CMPIStringFT.release()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `str` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIStringFT.getCharPtr()

NAME

`CMPIStringFT.getCharPtr()` – get a pointer to a C-language string representation of a String

SYNOPSIS

```
const char* CMPIStringFT.getCharPtr(
    const CMPIString* str,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIStringFT.getCharPtr()` function shall return a char pointer to an internal character string representation of the `CMPISTring` structure.

The `str` argument is a pointer to a `CMPIString` structure.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIStringFT.getCharPtr()` function shall return a character pointer to a C-language string. NULL shall be returned in case an error is detected.

This pointer must not be freed by the MI using `free()`. Instead, `CMPIStringFT.release()` will free the areas.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `str` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

6.3 CMPIArray Support

CMPIArrays encapsulate arrays of values of the same base types; however, some of them can be CIM NULL values. Property retrieval operations can return CMPIArray objects. MIs can produce CMPIArrays and use them in setProperty() operations. CMPIArrays are produced using a broker factory CMPIBrokerFT.newArray() function.

The CMPIArrayFT.getElementAt() and CMPIArrayFT.setElementAt() functions are used to retrieve and set individual array elements. Arrays shall be zero-based.

CMPIArray support is made available via the CMPIArrayFT function table:

```
typdef struct _CMPIArray {
    const void* hdl;
    const CMPIArrayFT* ft;
} CMPIArray;

typedef struct _CMPIArrayFT {
    int ftVersion;
    CMPIStatus (*release)
        (CMPIArray*);
    CMPIArray* (*clone)
        (const CMPIArray*,CMPIStatus*);
    CMPICount (*getSize)
        (const CMPIArray*,CMPIStatus*);
    CMPIType (*getSimpleType)
        (const CMPIArray*,CMPIStatus*);
    CMPIData (*getElementAt)
        (const CMPIArray*,CMPICount,CMPIStatus*);
    CMPIStatus (*setElementAt)
        (CMPIArray*,CMPICount,const CMPIValue*,CMPIType);
} CMPIArrayFT;
```

CMPIArray support is provided by the following functions.

CMPIArrayFT.clone()

NAME

`CMPIArrayFT.clone()` – create an independent copy of a `CMPIArray` object

SYNOPSIS

```
CMPIArray* CMPIArrayFT.clone(
    const CMPIArray* ar,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIArrayFT.clone()` function shall create an independent copy of a `CMPIArray` object.

The `ar` argument points to the `CMPIArray` structure to be copied.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIArrayFT.clone()` function shall return a pointer to the copied `CMPIArray` object. NULL shall be returned in case an error is detected.

The resulting Array object must be explicitly released using `CMPIArrayFT.release()`.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `ar` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIArrayFT.release()

NAME

`CMPIArrayFT.release()` – indicate that the Array object will not be used any further and may be freed by the CMPI run-time system

SYNOPSIS

```
CMPIStatus CMPIArrayFT.release(
    CMPIArray* ar
);
```

DESCRIPTION

The `CMPIArrayFT.release()` function shall indicate that a `CMPIArray` object will not be used any further, and may be freed by the CMPI run-time system.

The `ar` argument points to a `CMPIArray` structure.

RETURN VALUE

The `CMPIArrayFT.release()` function shall return the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `ar` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIArrayFT.getElementAt()

NAME

`CMPIArrayFT.getElementAt()` – get an element value defined by its index

SYNOPSIS

```
CMPIData CMPIArrayFT.getElementAt(
    const CMPIArray* ar,
    CMPICount index,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIArrayFT.getElementAt()` function shall return an element of an array, which can be a CIM NULL value.

The `ar` argument points to a `CMPIArray` structure.

The `index` argument specifies the zero based position of the property value in the internal data array.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIArrayFT.getElementAt()` function shall return a `CMPIData` structure containing the value of the specified element. In case of error, `CMPIData.state` shall be set to `CMPI_noValue`.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_NO_SUCH_PROPERTY	`index` value out of range.
CMPI_RC_ERR_INVALID_HANDLE	The `ar` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIArrayFT.getSimpleType()

NAME

`CMPIArrayFT.getSimpleType()` – get `CMPIArray` element type

SYNOPSIS

```
CMPIType CMPIArrayFT.getSimpleType(
    const CMPIArray* ar,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIArrayFT.getSimpleType()` function shall return the type of the elements of a CMPI`Array` object.

The `ar` argument points to a `CMPIArray` structure.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIArrayFT.getSimpleType()` function shall return a `CMPIType` type containing the type of the `CMPIArray` elements. In case of error, `CMPI_null` shall be returned.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `ar` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIArrayFT.getSize()

NAME

`CMPIArrayFT.getSize()` – get the number of elements in a `CMPIArray` object

SYNOPSIS

```
CMPICount CMPIArrayFT.getSize(
    const CMPIArray* ar,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIArrayFT.getSize()` function shall return the number of elements in a `CMPIArray` object.

The `ar` argument points to a `CMPIArray` structure.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIArrayFT.getSize()` function shall return a `CMPICount` containing the number of the Array elements. In case of error, 0 shall be returned.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `ar` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIArrayFT.setElementAt()

NAME

`CMPIArrayFT.setElementAt()` – set an element defined by its index

SYNOPSIS

```
CMPIStatus CMPIArrayFT.setElementAt(
    CMPIArray* ar,
    CMPICount index,
    const CMPIValue* value,
    CMPIType type
);
```

DESCRIPTION

The `CMPIArrayFT.setElementAt()` function sets an element of a `CMPIArray`, which can be a CIM NULL value.

The `ar` argument points to a `CMPIArray` structure.

The `index` argument specifies the zero-based position of the element in the internal data array.

The `value` argument points to a `CMPIValue` structure containing the value to be assigned to the element.

The `type` argument must either be the simple base type of the array or `CMPI_null`.

RETURN VALUE

The `CMPIArrayFT.setElementAt()` function shall return a `CMPIStatus` structure containing the service return status. In case of error, the element shall be set to `CMPI_null`.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_NO_SUCH_PROPERTY	`index` value out of range.
CMPI_RC_ERR_TYPE_MISATCH	`type` does not correspond to the simple base type of `ar`.
CMPI_RC_ERR_INVALID_HANDLE	The `ar` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

6.4 CMPIEnumeration Support

`CMPIEnumerations` are not directly created by an MI. MIs indirectly create enumerations by using successive `CMPIResultFT.returnXXX()` calls during execution of one of the enumerating MI functions. MIs, however, can request the MB to generate enumerations of other objects. In that case a `CMPIEnumeration` is returned.

In general, this support allows iteration through a `CMPIEnumeration`.

`CMPIEnumeration` support is made available via the `CMPIEnumerationFT` function table:

```
typedef struct _CMPIEnumeration {
    const void* hdl;
    const CMPIEnumerationFT* ft;
} CMPIEnumeration;

typedef struct _CMPIEnumerationFT {
    int ftVersion;
    CMPIStatus (*release)
        (CMPIEnumeration*);
    CMPIEnumeration* (*clone)
        (const CMPIEnumeration*,CMPIStatus*);
    CMPIData (*getNext)
        (const CMPIEnumeration*,CMPIStatus*);
    CMPIBoolean (*hasNext)
        (const CMPIEnumeration*,CMPIStatus*);
    CMPIArray* (*toArray)
        (const CMPIEnumeration*,CMPIStatus*);
} CMPIEnumerationFT;
```

`CMPIEnumeration` support is provided by the following functions.

CMPIEnumerationFT.clone()

NAME

`CMPIEnumerationFT.clone()` – create an independent copy of an Enumeration object

SYNOPSIS

```
CMPIEnumeration* CMPIEnumerationFT.clone(
    const CMPIEnumeration* en,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIEnumerationFT.clone()` function shall create an independent copy of an Enumeration object.

The `en` argument points to the `CMPIEnumeration` structure to be copied.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

The resulting Enumeration object must be explicitly released.

RETURN VALUE

The `CMPIEnumerationFT.clone()` function shall return a pointer to the copied Enumeration object. NULL shall be returned in case an error is detected.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `en` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIEnumerationFT.release()

NAME

`CMPIEnumerationFT.release()` – the Enumeration object will not be used any further and may be freed by the CMPI run-time system

SYNOPSIS

```
CMPIStatus CMPIEnumerationFT.release(
    CMPIEnumeration* en
);
```

DESCRIPTION

The `CMPIEnumerationFT.release()` function shall indicate that an Enumeration object will not be used any further, and may be freed by the CMPI run-time system.

The `en` argument points to a `CMPIEnumeration` structure.

RETURN VALUE

The `CMPIEnumerationFT.release()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `en` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIEnumerationFT.getNext()

NAME

`CMPIEnumerationFT.getNext()` – get the next element for a `CMPIEnumeration`

SYNOPSIS

```
CMPIData CMPIEnumerationFT.getNext(
    const CMPIEnumeration* en,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIEnumerationFT.getNext()` function shall return the next element for an Enumeration.

The `en` argument points to the `CMPIEnumeration` structure.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIEnumerationFT.getNext()` function shall return a `CMPIData` structure containing the element. In case of error, `CMPIData.state` shall be set to `CMPI_noValue`.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_NO_MORE_ELEMENTS	No more elements in `en`.
CMPI_RC_ERR_INVALID_HANDLE	The `en` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

`CMPIEnumerationFT.hasNext()`

CHANGE HISTORY

None.

CMPIEnumerationFT.hasNext()

NAME

`CMPIEnumerationFT.hasNext()` – test for any elements left in a `CMPIEnumeration`

SYNOPSIS

```
CMPIBoolean CMPIEnumerationFT.hasNext(
    const CMPIEnumeration* en,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIEnumerationFT.hasNext()` function shall test for any elements remaining in an Enumeration.

The `en` argument points to the `CMPIEnumeration` structure.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIEnumerationFT.hasNext()` function shall return a `CMPIBoolean` structure which is true if the Enumeration has elements.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `en` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIEnumerationFT.toArray()

NAME

`CMPIEnumerationFT.toArray()` – convert a `CMPIEnumeration` to a `CMPIArray`

SYNOPSIS

```
CMPIArray* CMPIEnumerationFT.toArray(
    const CMPIEnumeration* en,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIEnumerationFT.toArray()` function converts a `CMPIEnumeration` structure into a `CMPIArray` structure.

The `en` argument points to the `CMPIEnumeration` structure.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIEnumerationFT.toArray()` function shall return a pointer to a `CMPIArray` structure containing the elements from the Enumeration. In case of error, NULL shall be returned.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `en` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

6.5 CMPIInstance Support

CMPIInstance support is made available via the CMPIInstanceFT function table:

```
typedef struct _CMPIInstance {
    const void* hdl;
    const CMPIInstanceFT* ft;
} CMPIInstance;

typedef struct _CMPIInstanceFT {
    int ftVersion;
    CMPIStatus (*release)
        (CMPIInstance*);
    CMPIInstance* (*clone)
        (const CMPIInstance*,CMPIStatus*);
    CMPIData (*getProperty)
        (const CMPIInstance*,const char*, CMPIStatus*);
    CMPIData (*getPropertyAt)
        (const CMPIInstance*,CMPICount, CMPIString**,
         CMPIStatus*);
    CMPICount (*getPropertyCount)
        (const CMPIInstance*,CMPIStatus*);
    CMPIStatus (*setProperty)
        (const CMPIInstance*,const char*,const CMPIValue*,
         CMPIType);
    CMPIObjectPath* (*getObjectPath)
        (const CMPIInstance*,CMPIStatus*);
    CMPIStatus (*setPropertyFilter)
        (const CMPIInstance*,const char**,const char**);
    CMPIStatus (*setObjectPath)
        (CMPIInstance*,const CMPIObjectPath*);
    CMPIStatus (*setPropertyWithOrigin)
        (const CMPIInstance*, const char*,
         const CMPIValue*, const CMPIType, const char*);
} CMPIInstanceFT;
```

CMPIInstance support is provided by the following functions.

CMPIInstanceFT.clone()

NAME

`CMPIInstanceFT.clone()` – create an independent copy of a `CMPIInstance` object

SYNOPSIS

```
CMPIInstance* CMPIInstanceFT.clone(
    const CMPIInstance* inst,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIInstanceFT.clone()` function shall create an independent copy of an Instance object.

The `inst` argument points to the `CMPIInstance` structure to be copied.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIInstanceFT.clone()` function shall return a pointer to the copied Instance object. In case of error, NULL shall be returned.

The resulting Instance object must be explicitly released.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `inst` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIInstanceFT.release()

NAME

`CMPIInstanceFT.release()` – indicate that the `CMPIInstance` object will no longer be used and may be freed by the CMPI run-time system

SYNOPSIS

```
CMPIStatus CMPIInstanceFT.release(
    CMPIInstance* inst
);
```

DESCRIPTION

The `CMPIInstanceFT.release()` function shall indicate that an Instance object will not be used any further, and may be freed by the CMPI run-time system.

The `inst` argument points to a `CMPIInstance` structure.

RETURN VALUE

The `CMPIInstanceFT.release()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `inst` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIInstanceFT.getObjectPath()

NAME

`CMPIInstanceFT.getObjectPath()` – generate a `CMPIObjectPath` from the namespace, classname, and key properties of a `CMPIInstance`

SYNOPSIS

```
CMPIObjectPath* CMPIInstanceFT.getObjectPath(
    const CMPIInstance* inst,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIInstanceFT.getObjectPath()` function shall return a `CMPIObjectPath` generated from the namespace, classname, and key properties of a `CMPIInstance`.

The `inst` argument is a pointer to the `CMPIInstance` structure.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIInstanceFT.getObjectPath()` function shall return a pointer to a `CMPIObjectPath` structure. In case of error, NULL shall be returned.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `inst` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIInstanceFT.getProperty()

NAME

`CMPIInstanceFT.getProperty()` – get a named Property value

SYNOPSIS

```
CMPIData CMPIInstanceFT.getProperty(
    const CMPIInstance* inst,
    const char* name,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIInstanceFT.getProperty()` function shall return a named Property value.

The `inst` argument is a pointer to the `CMPIInstance` structure.

The `name` argument is a string containing the property name.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIInstanceFT.getProperty()` function shall return a `CMPIData` structure containing the named Property value. In case of error, `CMPIData.state` shall be set to `CMPI_noValue`.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_NO_SUCH_PROPERTY	Property not found.
CMPI_RC_ERR_INVALID_HANDLE	The `inst` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIInstanceFT.getPropertyAt()

NAME

`CMPIInstanceFT.getPropertyAt()` – get a Property value defined by its index

SYNOPSIS

```
CMPIData CMPIInstanceFT.getPropertyAt(
    const CMPIInstance* inst,
    CMPICount index,
    CMPIString** name,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIInstanceFT.getPropertyAt()` function shall return a Property value defined by its index.

The `inst` argument is a pointer to the `CMPIInstance` structure.

The `index` argument contains the zero-based index number of the Property in the internal data array.

The `name` output argument, if not NULL, is a pointer to a pointer to a `CMPIString` structure which is updated to contain the property name.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIInstanceFT.getProperyAt()` function shall return a `CMPIData` structure containing the Property value. In case of error, `CMPIData.state` shall be set to `CMPI_noValue`.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_NO_SUCH_PROPERTY	Property not found.
CMPI_RC_ERR_INVALID_HANDLE	The `inst` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIInstanceFT.getPropertyCount()

NAME

`CMPIInstanceFT.getPropertyCount()` – get the count of Properties contained in an Instance

SYNOPSIS

```
CMPICount CMPIInstanceFT.getPropertyCount(
    const CMPIInstance* inst,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIInstanceFT.getPropertyAt()` function shall return a `CMPICount` of Properties contained in an Instance.

The `inst` argument is a pointer to the `CMPIInstance` structure.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIInstanceFT.getProperyAt()` function shall return the count of Properties in the Instance. In case of error, 0 shall be returned.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `inst` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIInstanceFT.setProperty()

NAME

`CMPIInstanceFT.setProperty()` – add/replace a named Property

SYNOPSIS

```
CMPIStatus CMPIInstanceFT.setProperty(
    const const CMPIInstance* inst,
    const char* name,
    const CMPIValue* value,
    const CMPIType type
);
```

DESCRIPTION

The `CMPIInstanceFT.setProperty()` function shall add or replace a named property within an Instance. This function does not perform any semantic checking.

The `inst` argument is a pointer to the `CMPIInstance` structure.

The `name` argument is a string containing the Property name.

The `value` argument points to a `CMPIValue` structure containing the value to be assigned to the Property.

The `type` argument is a `CMPIType` structure defining the type of the value.

RETURN VALUE

The `CMPIInstanceFT.setProperty()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_TYPE_MISATCH	`type` does not correspond to class-defined type.
CMPI_RC_ERR_INVALID_HANDLE	The `inst` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIInstanceFT.setPropertyFilter()

NAME

`CMPIInstanceFT.setPropertyFilter()` – attach a property filter to an instance

SYNOPSIS

```
CMPIStatus CMPIInstanceFT.setPropertyFilter(
    const CMPIInstance* inst,
    const char** propertyList,
    const char** keyList
);
```

DESCRIPTION

The `CMPIInstanceFT.setPropertyFilter()` function shall attach a property filter to an instance. Subsequent `CMPIInstance.setProperty()` operations will consult the filter to determine whether a property is actually accepted. Previous filter settings for this instance are discarded. The MB may not support this function.

The `inst` argument points to a `CMPIInstance` to be tested.

The `propertyList` argument defines the properties that will be accepted by subsequent `setProperty()` operations. The `propertyList` argument is an array of pointers to character strings, terminated by a NULL pointer. A NULL value effectively means that all properties will be accepted. A pointer to an empty list means that no properties will be accepted.

The `keyList` argument shall be ignored by the MB; it is here to maintain binary compatibility with previous specifications. Providers should explicitly set the key names and values via the `CMPIInstanceMI.setObjectPath()` function.

RETURN VALUE

The `CMPIInstanceFT.setPropertyFilter()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_PARAMETER	The `keyList` argument is missing.
CMPI_RC_ERR_INVALID_HANDLE	The `inst` handle is invalid.
CMPI_RC_ERR_NOT_SUPPORTED	The MB does not support this function.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIInstanceFT.setPropertyWithOrigin()

NAME

`CMPIInstanceFT.setPropertyWithOrigin()` – add/replace a named Property value and origin

SYNOPSIS

```
CMPIStatus CMPIInstanceFT.setPropertyWithOrigin(
    const CMPIInstance* inst,
    const char* name,
    const CMPIValue* value,
    const CMPIType type,
    const char* origin
);
```

DESCRIPTION

The `CMPIInstanceFT.setPropertyWithOrigin()` function shall add or replace a named property within an Instance. This function does not perform any semantic checking.

The `inst` argument is a pointer to the `CMPIInstance` structure.

The `name` argument is a string containing the Property name.

The `value` argument points to a `CMPIValue` structure containing the value to be assigned to the Property.

The `type` argument is a `CMPIType` structure defining the type of the value.

The `origin` value specifies the instance origin. If NULL, then no origin is attached to the property.

RETURN VALUE

The `CMPIInstanceFT.setPropertyWithOrigin()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_TYPE_MISATCH	`type` does not correspond to class-defined type.
CMPI_RC_ERR_INVALID_HANDLE	The `inst` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

Specification for the Representation of CIM in XML (DSP0201), Section 3.1.4 for details on `ClassOrigin`

CHANGE HISTORY

None.

CMPIInstanceFT.setObjectPath()

NAME

`CMPIInstanceFT.setObjectPath()` – set/replace the `ObjectPath` component with an Instance

SYNOPSIS

```
CMPIStatus CMPIInstanceFT.setObjectPath(
    CMPIInstance* inst,
    const CMPIObjectPath* op
);
```

DESCRIPTION

The `CMPIInstanceFT.setObjectPath()` function shall set the `CMPIObjectPath` component of a `CMPIInstance`.

The `inst` argument points to a `CMPIInstance` structure containing a complete instance.

The `op` argument points to a `CMPIObjectPath` structure. This `objectpath` shall contain the namespace, classname, as well as all keys for the specified instance.

RETURN VALUE

The `CMPIInstanceFT.setObjectPath()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `inst` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

6.6 CMPIObjectPath Support

CMPIObjectPath support is made available via the CMPIObjectPathFT function table:

```
typedef struct _CMPIObjectPath {
    const void* hdl;
    const CMPIObjectPathFT* ft;
} CMPIObjectPath;

typedef struct _CMPIObjectPathFT {
    int ftVersion;
    CMPIStatus (*release)
        (CMPIObjectPath*);
    CMPIObjectPath* (*clone)
        (const CMPIObjectPath*,CMPIStatus*);
    CMPIStatus (*setNameSpace)
        (CMPIObjectPath*,const char*);
    CMPIString* (*getNameSpace)
        (const CMPIObjectPath*,CMPIStatus*);
    CMPIStatus (*setHostName)
        (CMPIObjectPath*,const char*);
    CMPIString* (*getHostName)
        (const CMPIObjectPath*,CMPIStatus*);
    CMPIStatus (*setClassName)
        (CMPIObjectPath*,const char*);
    CMPIString* (*getClassName)
        (const CMPIObjectPath*,CMPIStatus*);
    CMPIStatus (*addKey)
        (CMPIObjectPath*,const char*,
         const CMPIValue*,const CMPIType);
    CMPIData (*getKey)
        (const CMPIObjectPath*,const char*,CMPIStatus*);
    CMPIData (*getKeyAt)
        (const CMPIObjectPath*,CMPICount, CMPIString**,
         CMPIStatus*);
    CMPICount (*getKeyCount)
        (const CMPIObjectPath*,CMPIStatus*);
    CMPIStatus (*setNameSpaceFromObjectPath)
        (CMPIObjectPath* opThis,const CMPIObjectPath* src);
    CMPIStatus (*setHostAndNameSpaceFromObjectPath)
        (CMPIObjectPath* opThis,const CMPIObjectPath* src);

/* optional qualifier support */

    CMPIData (*getClassQualifier)
        (const CMPIObjectPath*,const char*,CMPIStatus*);
    CMPIData (*getPropertyQualifier)
        (const CMPIObjectPath*,const char*,
         const char*,CMPIStatus*);
    CMPIData (*getMethodQualifier)
        (const CMPIObjectPath*,const char*,
```

```
          const char*,CMPIStatus*);
    CMPIData (*getParameterQualifier)
        (const CMPIObjectPath*,const char*,const char*,
         const char*,CMPIStatus*);

/* end of optional qualifier support */

    CMPIString* (*toString)
        (const CMPIObjectPath*,CMPIStatus*);
} CMPIObjectPathFT;
```

`CMPIObjectPath` support is provided by the following functions.

CMPIObjectPathFT.clone()

NAME

`CMPIObjectPathFT.clone()` – create an independent copy of a CMPI `ObjectPath` object

SYNOPSIS

```
CMPIObjectPath* CMPIObjectPathFT.clone(
    const CMPIObjectPath* op,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIObjectPathFT.clone()` function shall create an independent copy of an `ObjectPath` object.

The `op` argument points to the `CMPIObjectPath` structure to be copied.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIObjectPathFT.clone()` function shall return a pointer to the copied `ObjectPath` object. In case of error, NULL shall be returned.

The resulting `CMPIObjectPath` object must be explicitly released.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `op` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIObjectPathFT.release()

NAME

`CMPIObjectPathFT.release()` – indicate that the `ObjectPath` object will no longer be used and may be freed by the CMPI run-time system

SYNOPSIS

```
CMPIStatus CMPIObjectPathFT.release(
    CMPIObjectPath* op
);
```

DESCRIPTION

The `CMPIObjectPathFT.release()` function shall indicate that an `ObjectPath` object will no longer be used, and may be freed by the CMPI run-time system.

The `op` argument points to a `CMPIObjectPath` structure.

RETURN VALUE

The `CMPIObjectPathFT.release()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `op` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIObjectPathFT.addKey()

NAME

`CMPIObjectPathFT.addKey()` – add/replace a named key property

SYNOPSIS

```
CMPIStatus CMPIObjectPathFT.addKey(
    CMPIObjectPath* op,
    const char* key,
    const CMPIValue* value,
    const CMPIType type
);
```

DESCRIPTION

The `CMPIObjectPathFT.addKey()` function shall add/replace a named key property.

The `op` argument points to a `CMPIObjectPath` structure.

The `key` argument points to a string containing the key property name.

The `value` argument is a pointer to a `CMPIValue` structure containing the value to be assigned to the key property.

The `type` argument is a `CMPIType` structure defining the type of the value to be assigned.

RETURN VALUE

The `CMPIObjectPathFT.addKey()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `op` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIObjectPathFT.getClassName()

NAME

`CMPIObjectPathFT.getClassName()` – get the classname component

SYNOPSIS

```
CMPIString* CMPIObjectPathFT.getClassName(
    const CMPIObjectPath* op,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIObjectPathFT.getClassName()` function shall get the classname component of an `ObjectPath`.

The `op` argument points to a `CMPIObjectPath` structure.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIObjectPathFT.getClassName()` function shall return a pointer to a `CMPIString` structure containing the classname component.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `op` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIObjectPathFT.getHostname()

NAME

`CMPIObjectPathFT.getHostname()` – get the hostname component

SYNOPSIS

```
CMPIString* CMPIObjectPathFT.getHostname(
    const CMPIObjectPath* op,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIObjectPathFT.getHostname()` function shall get the hostname component of an `ObjectPath`.

The `op` argument points to a `CMPIObjectPath` structure.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIObjectPathFT.getHostname()` function shall return a pointer to a `CMPIString` structure containing the hostname component.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `op` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIObjectPathFT.getKey()

NAME

`CMPIObjectPathFT.getKey()` – get a named key property value

SYNOPSIS

```
CMPIData CMPIObjectPathFT.getKey(
    const CMPIObjectPath* op,
    const char* key,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIObjectPathFT.getKey()` function shall get a named key property value.

The `op` argument points to a `CMPIObjectPath` structure.

The `key` argument points to a string containing the key name.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIObjectPathFT.getKey()` function shall return a `CMPIData` structure containing the named key value. In case of error, `CMPIData.state` shall be set to `CMPI_noValue`.

CIM XML does not prescribe transportation of precise key value types. There are only four types that can be returned by this operation. They are part of the `CMPIType` definition and are as follows:

```
#define CMPI_keyInteger   (CMPI_sint64)
#define CMPI_keyString    (CMPI_string)
#define CMPI_keyBoolean   (CMPI_boolean)
#define CMPI_keyRef       (CMPI_ref)
```

In addition, the `CMPI_keyValue` flag of `CMPIValueState` is set to indicate that the value is emanating from a `CMPIObjectPath`.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_NO_SUCH_PROPERTY	Key not found.
CMPI_RC_ERR_INVALID_HANDLE	The `op` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIObjectPathFT.getKeyAt()

NAME

`CMPIObjectPathFT.getKeyAt()` – get a key property value defined by its index

SYNOPSIS

```
CMPIData CMPIObjectPathFT.getKeyAt(
    const CMPIObjectPath* copThis,
    CMPICount* index,
    CMPIString** name,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIObjectPathFT.getKeyAt()` function shall get a key property value defined by its index.

The `op` argument points to a `CMPIObjectPath` structure.

The `index` argument specifies the zero-based position of the key property value within the internal data array.

The `name` argument is a pointer to a `CMPIstring` structure which is updated with the name of the key property.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIObjectPathFT.getKeyAt()` function shall return a `CMPIData` structure containing the named key value. In case of error, `CMPIData.state` shall be set to `CMPI_noValue`.

CIM XML does not prescribe transportation of precise key value types. There are only four types that can be returned by this operation. They are part of the `CMPIType` definition and are as follows:

```
#define CMPI_keyInteger   (CMPI_sint64)
#define CMPI_keyString    (CMPI_string)
#define CMPI_keyBoolean   (CMPI_boolean)
#define CMPI_keyRef       (CMPI_ref)
```

In addition, the `CMPI_keyValue` flag of `CMPIValueState` is set to indicate that the value is emanating from a `CMPIObjectPath`.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_NO_SUCH_PROPERTY	Key not found.
CMPI_RC_ERR_INVALID_HANDLE	The `op` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIObjectPathFT.getKeyCount()

NAME

`CMPIObjectPathFT.getKeyCount()` – get the number of key properties

SYNOPSIS

```
CMPICount CMPIObjectPathFT.getKeyCount(
    const CMPIObjectPath* op,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIObjectPathFT.getKeyCount()` function shall get the number of key properties contained in an `ObjectPath`.

The `op` argument points to a `CMPIObjectPath` structure.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIObjectPathFT.getKeyCount()` function shall return a `CMPICount` containing the number of key properties. In case of error, 0 shall be returned.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `op` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIObjectPathFT.getNameSpace()

NAME

`CMPIObjectPathFT.getNameSpace()` – get the namespace component

SYNOPSIS

```
CMPIString* CMPIObjectPathFT.getNameSpace(
    const CMPIObjectPath* op,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIObjectPathFT.getNameSpace()` function shall get the namespace component of an `ObjectPath`.

The `op` argument points to a `CMPIObjectPath` structure.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIObjectPathFT.getNameSpace()` function shall return a pointer to a `CMPIString` structure containing the namespace component. In case of error, NULL shall be returned.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `op` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIObjectPathFT.setHostname()

NAME

`CMPIObjectPathFT.setHostname()` – set/replace the hostname component

SYNOPSIS

```
CMPIStatus CMPIObjectPathFT.setHostname(
    CMPIObjectPath* op,
    const char* hn
);
```

DESCRIPTION

The `CMPIObjectPathFT.setHostname()` function shall set/replace the hostname component of an `ObjectPath`.

The `op` argument points to a `CMPIObjectPath` structure.

The `hn` argument is a string containing the new hostname.

RETURN VALUE

The `CMPIObjectPathFT.setHostname()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `op` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIObjectPathFT.setNameSpace()

NAME

`CMPIObjectPathFT.setNameSpace()` – set the namespace component

SYNOPSIS

```
CMPIStatus CMPIObjectPathFT.setNameSpace(
    CMPIObjectPath* op,
    const char* ns
);
```

DESCRIPTION

The `CMPIObjectPathFT.setNameSpace()` function shall set/replace the namespace component of an `ObjectPath`.

The `op` argument points to a `CMPIObjectPath` structure.

The `ns` argument is a string containing the new namespace.

RETURN VALUE

The `CMPIObjectPathFT.setNameSpace()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `op` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIObjectPathFT.setClassName()

NAME

`CMPIObjectPathFT.setClassName()` – set/replace the classname component

SYNOPSIS

```
CMPIStatus CMPIObjectPathFT.setClassName(
    CMPIObjectPath* op,
    char* cn
);
```

DESCRIPTION

The `CMPIObjectPathFT.setClassName()` function shall set/replace the classname component of an `ObjectPath`.

The `op` argument points to a `CMPIObjectPath` structure.

The `cn` argument is a string containing the new classname.

RETURN VALUE

The `CMPIObjectPathFT.setClassName()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `op` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIObjectPathFT.setNameSpaceFromObjectPath()

NAME

`CMPIObjectPathFT.setNameSpaceFromObjectPath()` – set/replace the namespace and classname components from an `ObjectPath`

SYNOPSIS

```
CMPIStatus CMPIObjectPathFT.setNameSpaceFromObjectPath(
    CMPIObjectPath* op,
    const CMPIObjectPath* src
);
```

DESCRIPTION

The `CMPIObjectPathFT.setNameSpaceFromObjectPath()` function shall set/replace the namespace and classname components from an `ObjectPath`.

The `op` argument points to a `CMPIObjectPath` structure to be modified.

The `src` argument points to a `CMPIObjectPath` structure used as the source for the new namespace and classname components.

RETURN VALUE

The `CMPIObjectPathFT.setNameSpaceFromObjectPath()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `op` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIObjectPathFT.setHostAndNameSpaceFromObjectPath()

NAME

`CMPIObjectPathFT.setHostAndNameSpaceFromObjectPath()` – set/replace the hostname, namespace, and classname components from an `ObjectPath`

SYNOPSIS

```
CMPIStatus CMPIObjectPathFT.setHostAndNameSpaceFromObjectPath(
    CMPIObjectPath* op,
    const CMPIObjectPath* src
);
```

DESCRIPTION

The `CMPIObjectPathFT.seHostAndtNameSpaceFromObjectPath()` function shall set/replace the hostname, namespace, and classname components from an `ObjectPath`.

The `op` argument points to a `CMPIObjectPath` structure to be modified.

The `src` argument points to a `CMPIObjectPath` structure used as the source for the new hostname, namespace, and classname components.

RETURN VALUE

The `CMPIObjectPathFT.setHostAndNameSpaceFromObjectPath()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `op` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIObjectPathFT.getClassQualifier()

NAME

`CMPIObjectPathFT.getClassQualifier()` – get the class qualifier value

SYNOPSIS

```
CMPIData CMPIObjectPathFT.getClassQualifier(
    const CMPIObjectPath *op,
    const char *qName,
    CMPIStatus *rc
);
```

DESCRIPTION

The `CMPIObjectPathFT.getClassQualifier()` function shall get the class qualifier value.

The `op` argument points to a `CMPIObjectPath` structure.

The `qName` argument is a string containing the qualifier name.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIObjectPathFT.getClassQualifier()` function shall return a `CMPIData` structure containing the named qualifier value.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `op` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIObjectPathFT.getMethodQualifier()

NAME

`CMPIObjectPathFT.getMethodQualifier()` – get the method qualifier value

SYNOPSIS

```
CMPIData CMPIObjectPathFT.getMethodQualifier(
    const CMPIObjectPath* op,
    const char *methodName,
    const char *qName,
    CMPIStatus *rc
);
```

DESCRIPTION

The `CMPIObjectPathFT.getMethodQualifier()` function shall get the method qualifier value.

The `op` argument points to a `CMPIObjectPath` structure.

The `methodName` argument is a string containing the method name.

The `qName` argument is a string containing the qualifier name.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIObjectPathFT.getMethodQualifier()` function shall return a `CMPIData` structure containing the named qualifier value.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `op` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIObjectPathFT.getParameterQualifier()

NAME

`CMPIObjectPathFT.getParameterQualifier()` – get the parameter qualifier value

SYNOPSIS

```
CMPIData CMPIObjectPathFT.getParameterQualifier(
    const CMPIObjectPath* op,
    const char *mName,
    const char *pName,
    const char *qName,
    CMPIStatus *rc
);
```

DESCRIPTION

The `CMPIObjectPathFT.getParameterQualifier()` function shall get the parameter qualifier value.

The `op` argument points to a `CMPIObjectPath` structure.

The `mName` argument is a string containing the method name.

The `pName` argument is a string containing the parameter name.

The `qName` argument is a string containing the qualifier name.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIObjectPathFT.getParameterQualifier()` function shall return a `CMPIData` structure containing the named qualifier value.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `op` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIObjectPathFT.getPropertyQualifier()

NAME

`CMPIObjectPathFT.getPropertyQualifier()` – get the property qualifier value

SYNOPSIS

```
CMPIData CMPIObjectPathFT.getPropertyQualifier(
    const CMPIObjectPath* op,
    const char *pName,
    const char *qName,
    CMPIStatus *rc
);
```

DESCRIPTION

The `CMPIObjectPathFT.getPropertyQualifier()` function shall get the property qualifier value.

The `op` argument points to a `CMPIObjectPath` structure.

The `pName` argument is a string containing the property name.

The `qName` argument is a string containing the qualifier name.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIObjectPathFT.getPropertyQualifier()` function shall return a `CMPIData` structure containing the named qualifier value.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `op` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

6.7 CMPIArgs Support

`CMPIArgs` is a container used to capture arguments for `invokeMethod()` functions.

`CMPIArgs` support is made available via the `CMPIArgsFT` function table:

```
typedef struct _CMPIArgs {
    const void* hdl;
    CMPIArgsFT* ft;
} CMPIArgs;

typedef struct _CMPIArgsFT{
    int ftVersion;
    CMPIStatus (*release)
        (CMPIArgs*);
    CMPIArgs* (*clone)
        (const CMPIArgs*,CMPIStatus*);
    CMPIStatus (*addArg)
        (CMPIArgs*,const char*,const CMPIValue*,
         const CMPIType);
    CMPIData (*getArg)
        (const CMPIArgs*,const char*,CMPIStatus*);
    CMPIData (*getArgAt)
        (const CMPIArgs*,CMPICount,CMPIString**,CMPIStatus*);
    CMPICount (*getArgCount)
        (const CMPIArgs*,CMPIStatus*);
} CMPIArgsFT;
```

`CMPIArgs` support is provided by the following functions.

CMPIArgsFT.clone()

NAME

`CMPIArgsFT.clone()` – create an independent copy of a `CMPIArgs` object

SYNOPSIS

```
CMPIArgs* CMPIArgsFT.clone(
    const CMPIArgs* args,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIArgsFT.clone()` function shall create an independent copy of a `CMPIArgs` object.

The `args` argument points to the `CMPIArgs` structure to be copied.

The `rc` argument points to a `CMPIStatus` structure containing the service return status.

The resulting `Args` object must be explicitly released.

RETURN VALUE

The `CMPIArgsFT.clone()` function shall return a pointer to the copied `CMPIArgs` object. In case of error, NULL shall be returned.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `args` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIArgsFT.release()

NAME

`CMPIArgsFT.release()` – the `Args` object will not be used any further and may be freed by the CMPI run-time system

SYNOPSIS

```
CMPIStatus CMPIArgsFT.release(
    CMPIArgs* args
);
```

DESCRIPTION

The `CMPIArgsFT.release()` function shall indicate that an `Args` object will not be used any further, and may be freed by the CMPI run-time system.

The `args` argument points to a `CMPIArgs` structure.

RETURN VALUE

The `CMPIArgsFT.release()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `args` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIArgsFT.addArg()

NAME

`CMPIArgsFT.addArg()` – add/replace a named argument

SYNOPSIS

```
CMPIStatus CMPIArgsFT.addArg(
    CMPIArgs* args,
    const char* name,
    const CMPIValue* value,
    const CMPIType type
);
```

DESCRIPTION

The `CMPIArgsFT.addArg()` function shall add or replace a named argument within an `Args` structure.

The `args` argument points to a `CMPIArgs` structure.

The `name` argument is a string containing the name of the argument to be added or replaced.

The `value` argument points to a `CMPIValue` structure containing the value to be assigned to the argument.

The `type` argument identifies the type of the argument.

The following types are supported:

```
CMPIBoolean
CMPIChar16
CMPIUint8
CMPIUint16
CMPIUint32
CMPIUint64
CMPISint8
CMPISint16
CMPISint32
CMPISint64
CMPIReal32
CMPIReal64
CMPIString*
CMPIObjectPath*
```

RETURN VALUE

The `CMPIArgsFT.addArg()` function shall return a `CMPIStatus` structure containing the service return status. In case of error, `CMPIData.state` shall be set to `CMPI_noValue`.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_DATA_TYPE	Data type not valid.
CMPI_RC_ERR_INVALID_HANDLE	The `args` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIArgsFT.getArg()

NAME

`CMPIArgsFT.getArg()` – get a named argument value

SYNOPSIS

```
CMPIData CMPIArgsFT.getArg(
    const CMPIArgs* args,
    const char* name,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIArgsFT.getArg()` function shall return the value of a named argument.

The `args` argument points to a `CMPIArgs` structure.

The `name` argument is a string containing the name of the argument.to be retrieved.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIArgsFT.getArg()` function shall return a `CMPIData` structure containing the value of the named argument.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIArgsFT.getArgAt()

NAME

`CMPIArgsFT.getArgAt()` – get an argument value defined by its index

SYNOPSIS

```
CMPIData CMPIArgsFT.getArgAt(
    const CMPIArgs* args,
    CMPICount index,
    CMPIString** name,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIArgsFT.getArgAt()` function shall return the value of an argument defined by its index in the internal data array.

The `args` argument points to a `CMPIArgs` structure.

The `index` argument specifies the zero-based position of the argument in the internal data array.

The `name` argument, if not NULL, points to a pointer to a `CMPIString` structure used to return the argument name.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIArgsFT.getArgAt()` function shall return a `CMPIData` structure containing the value of the specified argument. In case of error, `CMPIData.state` shall be set to `CMPI_noValue`.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_NO_SUCH_PROPERTY	Argument not found.
CMPI_RC_ERR_INVALID_HANDLE	The `args` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIArgsFT.getArgCount()

NAME

`CMPIArgsFT.getArgCount()` – get the number of arguments contained in a `CMPIArgs` structure

SYNOPSIS

```
CMPICount CMPIArgsFT.getArgCount(
    const CMPIArgs* args,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIArgsFT.getArgCount()` function gets the number of arguments contained in a `CMPIArgs` structure.

The `args` argument points to a `CMPIArgs` structure.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIArgsFT.getArgCount()` function shall return a `CMPICount` containing the number of arguments contained in the `Args` structure. In case of error, 0 shall be returned.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `args` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

6.8 CMPIDateTime Support

In order to be platform-independent, `CMPIDateTime` is implemented as an encapsulated type. It supports two ways of expressing time in binary form using a `long long` C type and as a `CMPIString` using CIM `datetime` fixed string format. Time can be set by asking for the current time of day or by using any of the two formats defined before as input. `CMPIDateTime` supports the UTC notion of interval *versus* time of date values.

`CMPIDateTime` support is made available via the `CMPIDateTimeFT` function table:

```
typedef struct _CMPIDateTime {
    const void* hdl;
    const CMPIDateTimeFT* ft;
} CMPIDateTime;

typedef struct _CMPIDateTimeFT {
    int ftVersion;
    CMPIStatus (*release)
        (CMPIDateTime*);
    CMPIDateTime* (*clone)
        (const CMPIDateTime*,CMPIStatus*);
    CMPIUint64 (*getBinaryFormat)
        (const CMPIDateTime*,CMPIStatus*);
    CMPIString* (*getStringFormat)
        (const CMPIDateTime*,CMPIStatus*);
    CMPIBoolean (*isInterval)
        (const CMPIDateTime*,CMPIStatus*);
} CMPIDateTimeFT;
```

`CMPIDateTime` support is provided by the following functions.

CMPIDateTimeFT.clone()

NAME

`CMPIDateTimeFT.clone()` – create an independent copy of a `DateTime` object

SYNOPSIS

```
CMPIDateTime* CMPIDateTimeFT.clone(
    const CMPIDateTime* dt,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIDateTimeFT.clone()` function shall create an independent copy of a `DateTime` object.

The `dt` argument points to the `CMPIDateTime` structure to be copied.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

The resulting `DateTime` object must be explicitly released.

RETURN VALUE

The `CMPIDateTimeFT.clone()` function shall return a pointer to the copied `DateTime` object. In case of error, NULL shall be returned.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `dt` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIDateTimeFT.release()

NAME

`CMPIDateTimeFT.release()` – the `DateTime` object will not be used any further and may be freed by the CMPI run-time system

SYNOPSIS

```
CMPIStatus CMPIDateTimeFT.release(
    CMPIDateTime* dt
);
```

DESCRIPTION

The `CMPIDateTimeFT.release()` function shall indicate that a `DateTime` object will not be used any further, and may be freed by the CMPI run-time system.

The `dt` argument points to a `CMPIDateTime` structure.

RETURN VALUE

The `CMPIDateTimeFT.release()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `dt` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIDateTimeFT.getBinaryFormat()

NAME

`CMPIDateTimeFT.getBinaryFormat()` – get a `DateTime` value in binary format

SYNOPSIS

```
CMPIUint64 CMPIDateTimeFT.getBinaryFormat(
    const CMPIDateTime* dt,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIDateTimeFT.getBinaryFormat()` function shall return the value of a `DateTime` object as a 64-bit unsigned integer in microseconds starting since 00:00:00 GMT, Jan 1, 1970, or as an interval, depending on how the `CMPIDateTime` object was created.

The `dt` argument is a pointer the `CMPIDateTime` structure.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIDateTimeFT.getBinaryFormat()` function shall return a `CMPIUint64` structure containing the `DateTime` value in binary format. In case of error, 0 shall be returned.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `dt` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIDateTimeFT.getStringFormat()

NAME

`CMPIDateTimeFT.getStringFormat()` – get a `DateTime` value in UTC string format

SYNOPSIS

```
CMPIString* CMPIDateTimeFT.getStringFormat(
    const CMPIDateTime* dt,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIDateTimeFT.getStringFormat()` function shall return the value of a `DateTime` as a string formatted using the character format defined in the CIM XML v2 specification.

The `dt` argument is a pointer the `CMPIDateTime` structure.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIDateTimeFT.getStringFormat()` function shall return a pointer to a `CMPIString` structure containing the `DateTime` value in UTC string format. In case of error, NULL shall be returned.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `dt` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIDateTimeFT.isInterval()

NAME

`CMPIDateTimeFT.isInterval()` – test whether a `DateTime` object is an interval value

SYNOPSIS

```
CMPIBoolean CMPIDateTimeFT.isInterval(
    const CMPIDateTime* dt,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIDateTimeFT.isInterval()` function shall test whether a `DateTime` object is an interval value.

The `dt` argument is a pointer to the `CMPIDateTime` structure.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIDateTimeFT.isInterval()` function shall return a `CMPIBoolean` structure with the value `true` if the `DateTime` object is an interval value. In case of error, `false` shall be returned.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `dt` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

6.9 CMPISelectExp Support

CMPISelectExp support is made available via the CMPISelectExpFT function table:

```
typedef CMPIData CMPIAccessor(const char*,void*);

typedef struct _CMPISelectExp {
    const void* hdl;
    const CMPISelectExpFT* ft;
} CMPISelectExp;

typedef struct _CMPISelectExpFT {
    int ftVersion;
    CMPIStatus (*release)
        (CMPISelectExp*);
    CMPISelectExp* (*clone)
        (const CMPISelectExp*,CMPIStatus*);
    CMPIBoolean (*evaluate)
        (const CMPISelectExp*,const CMPIInstance*,
         CMPIStatus*);
    CMPIString* (*getString)
        (const CMPISelectExp*,CMPIStatus*);
    CMPISelectCond* (*getDOC)
        (const CMPISelectExp*,CMPIStatus*);
    CMPISelectCond* (*getCOD)
        (const CMPISelectExp*,CMPIStatus*);
    CMPIBoolean (*evaluateUsingAccessor)
        (const CMPISelectExp*,const CMPIAccessor*,
         void*,CMPIStatus*);
} CMPISelectExpFT;
```

CMPISelectExp support is provided by the following functions.

CMPISelectExpFT.clone()

NAME

`CMPISelectExpFT.clone()` – create an independent copy of a `SelectExp` object

SYNOPSIS

```
CMPISelectExp* CMPISelectExpFT.clone(
    const CMPISelectExp* se,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPISelectExpFT.clone()` function shall create an independent copy of a `SelectExp` object.

The `se` argument points to the `CMPISelectExp` structure to be copied.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

The resulting `SelectExp` object must be explicitly released.

RETURN VALUE

The `CMPISelectExpFT.clone()` function shall return a pointer to the copied `SelectExp` object. In case of error, NULL shall be returned.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `se` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPISelectExpFT.release()

NAME

`CMPISelectExpFT.release()` – the `SelectExp` object will not be used any further and may be freed by the CMPI run-time system

SYNOPSIS

```
CMPIStatus CMPISelectExpFT.release(
    CMPISelectExp* se
);
```

DESCRIPTION

The `CMPISelectExpFT.release()` function shall indicate that a `SelectExp` object will not be used any further, and may be freed by the CMPI run-time system.

The `se` argument points to a `CMPISelectExp` structure.

RETURN VALUE

The `CMPISelectExpFT.release()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `se` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPISelectExpFT.evaluate()

NAME

`CMPISelectExpFT.evaluate()` – evaluate an Instance using a Select Expression

SYNOPSIS

```
CMPIBoolean CMPISelectExpFT.evaluate(
    const CMPISelectExp* se,
    const CMPIInstance* inst,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPISelectExpFT.evaluate()` function shall evaluate an Instance using a Select Expression.

The `se` argument is a pointer to a `CMPISelectExp` structure containing the Select Expression.

The `inst` argument is a pointer to a `CMPIInstance` structure containing the Instance to be evaluated.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPISelectExpFT.evaluate()` function shall return a `CMPIBoolean` value indicating the result of the evaluation. In case the evaluation fails, `false` shall be returned.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `se` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPISelectExpFT.getCOD()

NAME

`CMPISelectExpFT.getCOD()` – return a selection as a conjunction of disjunctions

SYNOPSIS

```
CMPISelectCond* CMPISelectExpFT.getCOD(
    const CMPISelectExp* se,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPISelectExpFT.getCOD()` function shall return the selection as a conjunction of disjunctions. This function transforms the filter's WHERE clause into a canonical Conjunction of Disjunctions form (AND or OR'ed comparison expressions). This enables handling of the `where` expression more easily than using a tree form.

The `se` argument is a pointer to a `CMPISelectExp` structure containing the Select Expression.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

Support for this function is optional. Availability of this support is indicated by the `CMPI_MB_QueryNormalization` flag in `CMPIBrokerFT.brokerCapabilities`.

RETURN VALUE

The `CMPISelectExpFT.getCOD()` function shall return a pointer to a `CMPISelectCond` structure containing the transformed selection. In case of error, NULL shall be returned.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_NOT_SUPPORTED	`QueryNormalization` not supported.
CMPI_RC_ERR_INVALID_HANDLE	The `se` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPISelectExpFT.getDOC()

NAME

`CMPISelectExpFT.getDOC()` – return a selection as a disjunction of conjunctions

SYNOPSIS

```
CMPISelectCond* CMPISelectExpFT.getDOC(
    const CMPISelectExp* se,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPISelectExpFT.getDOC()` function shall return the selection as a disjunction of conjunctions. This function transforms the filter's WHERE clause into a canonical Disjunction of Conjunctions form (OR or AND'ed comparison expressions). This enables handling of the `where` expression more easily than using a tree form.

The `se` argument is a pointer to a `CMPISelectExp` structure containing the Select Expression.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

Support for this function is optional. Availability of this support is indicated by the `CMPI_MB_QueryNormalization` flag in `CMPIBrokerFT.brokerCapabilities`.

RETURN VALUE

The `CMPISelectExpFT.getDOC()` function shall return a pointer to a `CMPISelectCond` structure containing the transformed selection. In case of error, NULL shall be returned.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_NOT_SUPPORTED	`QueryNormalization` not supported.
CMPI_RC_ERR_INVALID_HANDLE	The `se` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPISelectExpFT.getString()

NAME

`CMPISelectExpFT.getString()` – return a Select Expression in string format

SYNOPSIS

```
CMPIString* CMPISelectExpFT.getString(
    const CMPISelectExp* se,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPISelectExpFT.getString()` function shall return the selection as a string.

The `se` argument is a pointer to a `CMPISelectExp` structure containing the Select Expression.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPISelectExpFT.getString()` function shall return a pointer to a `CMPIString` structure cointaining the Select Expression in string format. In case of error, NULL shall be returned.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `se` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPISelectExpFT.evaluateUsingAccessor()

NAME

`CMPISelectExpFT.evaluateUsingAccessor()` – evaluate using an accessor function

SYNOPSIS

```
CMPIBoolean CMPISelectExpFT.evaluateUsingAccessor(
    const CMPISelectExp* se,
    const CMPIAccessor* accessorFnc,
    void* parm,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPISelectExpFT.evaluateUsingAccessor()` function shall evaluate a select expression using a property data accessor function. This function is a variation of `CMPISelectExpFT.evaluate()` – it enables evaluation without the need to create a `CMPIInstance`.

The `se` argument is a pointer to a `CMPISelectExp` structure containing the Select Expression.

The `accessorFnc` argument is a pointer to a property value accessor function. The evaluation process will invoke this function to request a `CMPIData` structure for a particular property. The signature of the accessor function is:

```
CMPIData CMPIAccessor(const char* propertyName, void* parm);
```

The `parm` argument is a parameter that will be passed to the accessor function and can be used for providing context data to the accessor function.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPISelectExpFT.evaluateUsingAccessor()` function shall return a `CMPIBoolean` value indicating the result of the evaluation. In case the evaluation fails, false shall be returned.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `se` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

`CMPIPredicate.evaluateUsingAccessor()`

CHANGE HISTORY

None.

6.10 CMPISelectCond Support

`CMPISelectCond` support is optional. Availability of this support is indicated by the `CMPI_MB_QueryNormalization` flag in `CMPIBrokerFT.brokerCapabilities`.

`CMPISelectCond` support is made available via the `CMPISelectCondFT` function table:

```
typedef struct _CMPISelectCond {
    const void* hdl;
    CMPISelectCondFT* ft;
} CMPISelectCond;

typedef struct _CMPISelectCondFT {
    int ftVersion;
    CMPIStatus (*release)
        (CMPISelectCond*);
    CMPISelectCond* (*clone)
        (const CMPISelectCond*,CMPIStatus*);
    CMPICount (*getCountAndType)
        (const CMPISelectCond*,int*,CMPIStatus*);
    CMPISubCond* (*getSubCondAt)
        (const CMPISelectCond*,CMPICount,CMPIStatus*);
} CMPISelectCond;
```

`CMPISelectCondSupport` is provided by the following functions.

CMPISelectCondFT.clone()

NAME

`CMPISelectCondFT.clone()` – create an independent copy of a `SelectCond` object

SYNOPSIS

```
CMPISelectCond* CMPISelectCondFT.clone(
    const CMPISelectCond* sc,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPISelectCondFT.clone()` function shall create an independent copy of a `SelectCond` object.

The `sc` argument points to the `CMPISelectCond` structure to be copied.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

The resulting `SelectCond` object must be explicitly released.

RETURN VALUE

The `CMPISelectCondFT.clone()` function shall return a pointer to the copied `SelectCond` object. In case of error, NULL shall be returned.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_NOT_SUPPORTED	`QueryNormalization` not supported.
CMPI_RC_ERR_INVALID_HANDLE	The `sc` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPISelectCondFT.release()

NAME

`CMPISelectCondFT.release()` – the `SelectCond` object will not be used any further and may be freed by the CMPI run-time system

SYNOPSIS

```
CMPIStatus CMPISelectCondFT.release(
    CMPISelectCond* sc
);
```

DESCRIPTION

The `CMPISelectCondFT.release()` function shall indicate that a `SelectCond` object will not be used any further, and may be freed by the CMPI run-time system.

The `sc` argument points to a `CMPISelectCond` structure.

RETURN VALUE

The `CMPISelectCondFT.release()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_NOT_SUPPORTED	`QueryNormalization` not supported.
CMPI_RC_ERR_INVALID_HANDLE	The `sc` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPISelectCondFT.getCountAndType()

NAME

`CMPISelectCondFT.getCountAndType()` – return the number and type of subconditions that are part of a `SelectCond` object

SYNOPSIS

```
CMPICount CMPISelectCondFT.getCountAndType(
    const CMPISelectCond* sc,
    int* type,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPISelectCondFT.getCountAndType()` function shall return the count and type of the subconditions that are part of a `SelectCond` object.

The `sc` argument points to a `CMPISelectCond` structure.

The `type` output argument, if not NULL, is a pointer to an integer which is updated with the `SelectCond` type. A value of 0 indicates a DOC type, and a value of 1 indicates a COD type. If `type` is NULL, no type information is returned.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPISelectCondFT.getCountAndType()` function shall return a `CMPICount` containing the number of subconditions. In case of error, 0 shall be returned.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_NOT_SUPPORTED	`QueryNormalization` not supported.
CMPI_RC_ERR_INVALID_HANDLE	The `sc` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPISelectCondFT.getSubCondAt()

NAME

`CMPISelectCondFT.getSubCondAt()` – return a `SubCond` element based on its index

SYNOPSIS

```
CMPISubCond* CMPISelectCondFT.getSubCondAt(
    const CMPISelectCond* sc,
    CMPICount index,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPISelectCondFT.getSubCondAt()` function shall return the `SubCond` element at the specified index into the `sc` `SelectCond` object.

The `sc` argument points to a `CMPISelectCond` structure.

The `index` argument specifies the zero-based position of the `SubCond` element in the internal data array.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPISelectCondFT.getSubCondAt()` function shall return a pointer to a `CMPISubCond` structure containing the `SubCond` element at the position specified by the `index` parameter. In case of error, NULL shall be returned.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_NOT_SUPPORTED	`QueryNormalization` not supported.
CMPI_RC_ERR_NOT_SUCH_PROPERTY	`index` values out of bounds.
CMPI_RC_ERR_INVALID_HANDLE	The `sc` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

6.11 CMPISubCond Support

CMPISubCond support is optional. Availability of this support is indicated by the CMPI_MB_QueryNormalization flag in CMPIBrokerFT.brokerCapabilities.

CMPISubCond support is made available via the CMPISubCondFT function table:

```
typedef struct _CMPISubCond {
    const void* hdl;
    const CMPISubCondFT* ft;
} CMPISubCond;

typedef struct _CMPISubCondFT {
    int ftVersion;
    CMPIStatus (*release)
        (CMPISubCond*);
    CMPISubCond* (*clone)
        (const CMPISubCond*,CMPIStatus*);
    CMPICount (*getCount)
        (const CMPISubCond*,CMPIStatus*);
    CMPIPredicate* (*getPredicateAt)
        (const CMPISubCond*,CMPICount,CMPIStatus*);
    CMPIPredicate* (*getPredicate)
        (const CMPISubCond*,const char*,CMPIStatus*);
} CMPISubCondFT;
```

CMPISubCond support is provided by the following functions.

CMPISubCondFT.clone()

NAME

CMPISubCondFT.clone() – create an independent copy of a SubCond object

SYNOPSIS

```
CMPISubCond* CMPISubCondFT.clone(
    const CMPISubCond* sc,
    CMPIStatus* rc
);
```

DESCRIPTION

The CMPISubCondFT.clone() function shall create an independent copy of a SubCond object.

The sc argument points to the CMPISubCond structure to be copied.

The rc output argument, if not NULL, is used to return a CMPIStatus structure containing the service return status.

The resulting SubCond object must be explicitly released.

RETURN VALUE

The CMPISubCondFT.clone() function shall return a pointer to the copied SubCond object. In case of error, NULL shall be returned.

The following CMPIrc codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_NOT_SUPPORTED	QueryNormalization not supported.
CMPI_RC_ERR_INVALID_HANDLE	The sc handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPISubCondFT.release()

NAME

`CMPISubCondFT.release()` – the `SubCond` object will not be used any further and may be freed by the CMPI run-time system

SYNOPSIS

```
CMPIStatus CMPISubCondFT.release(
    CMPISubCond* sc
);
```

DESCRIPTION

The `CMPISubCondFT.release()` function shall indicate that a `SubCond` object will not be used any further, and may be freed by the CMPI run-time system.

The `sc` argument points to a `CMPISubCond` structure.

RETURN VALUE

The `CMPISubCondFT.release()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_NOT_SUPPORTED	`QueryNormalization` not supported.
CMPI_RC_ERR_INVALID_HANDLE	The `sc` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPISubCondFT.getCount()

NAME

`CMPISubCondFT.getCount()` – return the number of predicates that are part of a subcondition

SYNOPSIS

```
CMPICount CMPISubCondFT.getCount(
    const CMPISelectCond* sc,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPISubCondFT.getCount()` function shall return the number of predicates that are part of a subcondition.

The `sc` argument is a pointer to a `CMPISubCondFT` structure containing the subcondition.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPISubCondFT.getCount()` function shall return a `CMPICount` containing the number of predicates. In case of error, 0 shall be returned.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_NOT_SUPPORTED	`QueryNormalization` not supported.
CMPI_RC_ERR_INVALID_HANDLE	The `sc` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPISubCondFT.getPredicate()

NAME

`CMPISubCondFT.getPredicate()` – return a named predicate element

SYNOPSIS

```
CMPIPredicate* CMPISubCondFT.getPredicate(
    const CMPISubCond* sc,
    const char* name,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPISubCondFT.getPredicate()` function shall return a named predicate from a subcondition. This function treats the list of predicates as a list of named entries and enables getting a particular predicate by name.

The `sc` argument is a pointer to a `CMPISubCondFT` structure containing the subcondition.

The `name` argument is a string containing the predicate name. The name is the left-hand side of the predicate.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPISubCondFT.getPredicate()` function shall return a pointer to a `CMPIPredicate` structure containing the named predicate. In case of error, NULL shall be returned.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_NOT_SUPPORTED	`QueryNormalization` not supported.
CMPI_RC_ERR_INVALID_HANDLE	The `sc` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPISubCondFT.getPredicateAt()

NAME

`CMPISubCondFT.getPredicateAt()` – return a predicate based on its index

SYNOPSIS

```
CMPIPredicate* CMPISubCondFT.getPredicateAt(
    const CMPISubCond* sc,
    CMPICount index,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPISubCondFT.getPredicatAte()` function shall return a predicate from a subcondition based on its index.

The `sc` argument is a pointer to a `CMPISubCondFT` structure containing the subcondition.

The `index` argument specifies the zero-based index of the predicate in the internal data array.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPISubCondFT.getPredicate()` function shall return a pointer to a `CMPIPredicate` structure containing the predicate. In case of error, NULL shall be returned.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_NOT_SUPPORTED	`QueryNormalization` not supported.
CMPI_RC_ERR_NOT_SUCH_PROPERTY	`index` value out of bounds.
CMPI_RC_ERR_INVALID_HANDLE	The `sc` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

6.12 CMPIPredicate Support

`CMPIPredicate` support is optional. Availability of this support is indicated by the `CMPI_MB_QueryNormalization` flag in `CMPIBrokerFT.brokerCapabilities`.

`CMPIPredicate` support is made available via the `CMPIPredicateFT` function table:

```
typedef struct _CMPIPredicate {
    const void* hdl;
    const CMPIPredicateFT* ft;
} CMPIPredicate;

typedef struct _CMPIPredicateFT {
    int ftVersion;
    CMPIStatus (*release)(CMPIPredicate*);
    CMPIPredicate* (*clone)
        (const CMPIPredicate*,CMPIStatus*);
    CMPIStatus (*getData)
        (const CMPIPredicate*,CMPIType*,
         CMPIPredOp*,CMPIString**,CMPIString**);
    CMPIBoolean (*evaluateUsingAccessor)
        (const CMPIPredicate*,const CMPIAccessor*,
         void*,CMPIStatus*);
} CMPIPredicateFT;
```

`CMPIPredicate` support is provided by the following functions.

CMPIPredicateFT.clone()

NAME

`CMPIPredicateFT.clone()` – create an independent copy of a Predicate object

SYNOPSIS

```
CMPIPredicate* CMPIPredicateFT.clone(
    const CMPIPredicate* pr,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIPredicateFT.clone()` function shall create an independent copy of a Predicate object.

The `pr` argument points to the `CMPIPredicate` structure to be copied.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

The resulting Predicate object must be explicitly released.

RETURN VALUE

The `CMPIPredicateFT.clone()` function shall return a pointer to the copied Predicate object. In case of error, NULL shall be returned.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_NOT_SUPPORTED	`QueryNormalization` not supported.
CMPI_RC_ERR_INVALID_HANDLE	The `sc` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIPredicateFT.release()

NAME

`CMPIPredicateFT.release()` – the Predicate object will not be used any further and may be freed by the CMPI run-time system

SYNOPSIS

```
CMPIStatus CMPIPredicateFT.release(
    CMPIPredicate* pr
);
```

DESCRIPTION

The `CMPIPredicateFT.release()` function shall indicate that a Predicate object will not be used any further, and may be freed by the CMPI run-time system.

The `pr` argument points to a `CMPIPredicate` structure.

RETURN VALUE

The `CMPIPredicateFT.release()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_NOT_SUPPORTED	`QueryNormalization` not supported.
CMPI_RC_ERR_INVALID_HANDLE	The `sc` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIPredicateFT.getData()

NAME

`CMPIPredicateFT.getData()` – get the predicate components

SYNOPSIS

```
CMPIStatus CMPIPredicateFT.getData(
    const CMPIPredicate* pr,
    CMPIType* type,
    CMPIPredOp* op,
    CMPIString** lhs,
    CMPIString** rhs
);
```

DESCRIPTION

The `CMPIPredicateFT.getData()` function shall get the predicate components.

The `pr` argument points to a `CMPIPredicate` structure to be evaluated.

The `type` output argument, if not NULL, points to a `CMPIType` structure defining the property type.

The `op` output argument, if not NULL, points to a `CMPIPredOp` structure containing the predicate operation.

The `lhs` output argument, if not NULL, points to a `CMPIString` structure that is updated with the left-hand side of the predicate.

The `rhs` output argument, if not NULL, points to a `CMPIString` structure that is updated with the right-hand side of the predicate.

RETURN VALUE

The `CMPIPredicateFT.getData()` function shall return a `CMPIStatus` structure containing the service return status. In case of error, all output arguments shall be set to NULL.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_NOT_SUPPORTED	`QueryNormalization` not supported.
CMPI_RC_ERR_INVALID_HANDLE	The `sc` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIPredicateFT.evaluateUsingAccessor()

NAME

`CMPIPredicateFT.evaluateUsingAccessor()` – evaluate using an accessor function

SYNOPSIS

```
CMPIBoolean CMPIPredicateFT.evaluateUsingAccessor(
    const CMPIPredicate* pr,
    const CMPIAccessor* accessorFnc,
    void* parm,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIPredicateFT.evaluateUsingAccessor()` function shall evaluate a predicate using a property data accessor function.

The `pr` argument is a pointer to a `CMPIPredicate` structure containing a predicate expression.

The `accessorFnc` argument is a pointer to a property value accessor function. The evaluation process will invoke this function to request a `CMPIData` structure for a particular property. The signature of the accessor function is:

```
CMPIData CMPIAccessor(const char* propertyName, void* parm);
```

The `parm` argument is a parameter that will be passed to the accessor function and can be used for providing context data to the accessor function.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIPredicateFT.evaluateUsingAccessor()` function shall return a `CMPIBoolean` value indicating the result of the evaluation. In case the evaluation fails, false shall be returned.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_INVALID_HANDLE	The `se` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

`CMPISelectExpFT.evaluateUsingAccessor()`

CHANGE HISTORY

None.

6.13 CMPIError Support

`CMPIError` support is optional.

Availability of this support is indicated by the `CMPI_MB_Supports_Extended_Error` flag in `CMPIBrokerFT.brokerCapabilities`.

`CMPIError` support is made available via the `CMPIErrorFT` function table:

```
typedef struct CMPIError {
    void* hdl;
    CMPIErrorFT ft;
} CMPIError;

typedef struct _CMPIErrorFT {
    CMPISint32 ftVersion;
    CMPIStatus (*release)(CMPIError*);
    CMPIError* (*clone)(const CMPIError*, CMPIStatus*);
    CMPIErrorType (*getErrorType)(const CMPIError*,
        CMPIStatus*);
    CMPIString* (*getOtherErrorType)(const CMPIError*,
        CMPIStatus*);
    CMPIString* (*getOwningEntity)(const CMPIError*,
        CMPIStatus*);
    CMPIString* (*getMessageID)(const CMPIError*,
        CMPIStatus*);
    CMPIString* (*getMessage)(const CMPIError*, CMPIStatus*);
    CMPIErrorSeverity (*getPerceivedSeverity)(const
        CMPIError*, CMPIStatus*);
    CMPIErrorProbableCause (*getProbableCause)(
        const CMPIError*, CMPIStatus*);
    CMPIString* (*getProbableCauseDescription)(
        const CMPIError*, CMPIStatus*);
    CMPIArray* (*getRecommendedActions)(const CMPIError*,
        CMPIStatus*);
    CMPIString* (*getErrorSource)(const CMPIError*,
        CMPIStatus*);
    CMPIErrorSrcFormat (*getErrorSourceFormat)(
        const CMPIError*, CMPIStatus*);
    CMPIString* (*getOtherErrorSourceFormat)(const CMPIError*,
        CMPIStatus*);
    CMPIrc (*getCIMStatusCode)(const CMPIError*, CMPIStatus*);
    CMPIString* (*getCIMStatusCodeDescription)(
        const CMPIError*, CMPIStatus*);
    CMPIArray* (*getMessageArguments)(const CMPIError*,
        CMPIStatus*);
    CMPIStatus (*setErrorType)(CMPIError*, const
        CMPIErrorType);
    CMPIStatus (*setOtherErrorType)(CMPIError*, const char* );
    CMPIStatus (*setProbableCauseDescription)(CMPIError*,
        const char* );
    CMPIStatus (*setRecommendedActions)(CMPIError*,
```

```
            const CMPIArray*);
        CMPIStatus (*setErrorSource)(CMPIError*, const char*);
        CMPIStatus (*setErrorSourceFormat)(const CMPIError*,
            const CMPIErrorSrcFormat );
        CMPIStatus (*setOtherErrorSourceFormat)(CMPIError*,
            const char*);
        CMPIStatus (*setCIMStatusCodeDescription)(CMPIError*,
            const char*);
        CMPIStatus (*setMessageArguments)(CMPIError*, CMPIArray*);
    } CMPIErrorFT;
```

CMPIErrorFT. clone()

NAME

`CMPIErrorFT.clone()` – create an independent copy of a `CMPIError` object

SYNOPSIS

```
CMPIError* CMPIErrorFT.clone(
    const CMPIError* er,
    CMPIStatus* rc
);
```

DESCRIPTION

This function shall create an independent copy of a `CMPIError`.

The `er` argument is a pointer to the `CMPIError` object to be copied.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

This function shall return a pointer to a newly created `CMPIError` object. In case of error, NULL shall be returned.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation succeeded.
CMPI_RC_ERR_NOT_SUPPORTED	`CMPIError` is not supported.
CMPI_RC_ERR_FAILED	A generic error occurred.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIErrorFT.release()

NAME

`CMPIErrorFT.release()` – the `error` object will not be used any further and may be freed by the CMPI run-time system

SYNOPSIS

```
CMPIStatus CMPIErrorFT.release(
    CMPIError* er
);
```

DESCRIPTION

This function shall mark memory associated to a `CMPIError` object as no longer used.

The `er` argument points to a `CMPIError` object.

RETURN VALUE

Returns a status of the service return status. The following codes are valid:

CMPI_RC_OK	Indicates the `CMPIError` object was successfully created.
CMPI_RC_ERR_INVALID_PARAMETER	The `er` parameter is invalid.
CMPI_RC_ERR_NOT_SUPPORTED	`CMPIError` is not supported.
CMPI_RC_ERR_FAILED	A generic error occurred.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIErrorFT.getErrorType()

NAME

`CMPIErrorFT.getErrorType()` – get the type of the error

SYNOPSIS

```
CMPIErrorType getErrorType(
    CMPIError* er
    CMPIStatus* rc
);
```

DESCRIPTION

This function shall return the primary classification of the error that the `CMPIError` represents.

The `er` argument points to the `CMPIError` object that the error type will be taken from.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

This function shall return a `CMPIErrorType` enumeration which contains the error type.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation succeeded.
CMPI_RC_ERR_NOT_SUPPORTED	`CMPIError` is not supported.
CMPI_RC_ERR_FAILED	A generic error occurred.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

`CMPIErrorType`

CHANGE HISTORY

None.

CMPIErrorFT.getOtherErrorType()

NAME

`CMPIErrorFT.getOtherErrorType()` – get the alternative error type, if applicable

SYNOPSIS

```
CMPIString* getOtherErrorType(
    const CMPIError* er,
    CMPIStatus* rc
);
```

DESCRIPTION

This function shall return a free-form string describing the `ErrorType`, when the `ErrrorType` is set to `other`.

The `er` argument points to the `CMPIErro`r object that contains the information.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

This function shall return a pointer to a string which contains the data requested. This function must return NULL if the `ErrorType` is not set to `other`.

To see if this function failed you *must* inspect the `rc` argument.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation succeeded.
CMPI_RC_ERR_NOT_SUPPORTED	`CMPIError` is not supported.
CMPI_RC_ERR_FAILED	A generic error occurred.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

`getErrorType`

CHANGE HISTORY

None.

CMPIErrorFT.getOwningEntity()

NAME

`CMPIErrorFT.getOwningEntity()` – get a string which describes the owner

SYNOPSIS

```
CMPIString* getOwningEntity(
    const CMPIError* er,
    CMPIStatus* rc
);
```

DESCRIPTION

This function shall return the string which identifies the owner of the format of the Message described.

The `er` argument points to the `CMPIError` object that contains the information.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

This function shall return a pointer to a string which contains the data requested, or NULL on error.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation succeeded.
CMPI_RC_ERR_NOT_SUPPORTED	`CMPIError` is not supported.
CMPI_RC_ERR_FAILED	A generic error occurred.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIErrorFT.getMessageID()

NAME

`CMPIErrorFT.getMessageID()` – get a string which identifies the owning entity

SYNOPSIS

```
CMPIString* getMessageID(
    const CMPIError* er,
    CMPIStatus* rc
);
```

DESCRIPTION

This function shall return a pointer to a string which identifies the format of the message.

The `er` argument points to the `CMPIError` object that contains the information.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

This function shall return a pointer to a string which contains the data requested, or NULL on error.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation succeeded.
CMPI_RC_ERR_NOT_SUPPORTED	`CMPIError` is not supported.
CMPI_RC_ERR_FAILED	A generic error occurred.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIErrorFT.getMessage()

NAME

`CMPIErrorFT.getMessage()` – get a formatted message

SYNOPSIS

```
CMPIString* getMessage(
    const CMPIError* er,
    CMPIStatus* rc
);
```

DESCRIPTION

This function shall return a pointer to a string which describes the message. This message is constructed by applying the dynamic content of the message, described in `MessageArguments`, to the format string uniquely identified, within the scope of the `OwningEntity`, by `MessageID`.

The `er` argument points to the `CMPIError` object that contains the information.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

This function shall return a pointer to a string which contains the data requested, or NULL on error.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation succeeded.
CMPI_RC_ERR_NOT_SUPPORTED	`CMPIError` is not supported.
CMPI_RC_ERR_FAILED	A generic error occurred.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIErrorFT. getPerceivedSeverity()

NAME

`CMPIErrorFT.getPerceivedSeverity()` – get the severity of the error

SYNOPSIS

```
CMPIErrorSeverity getPerceivedSeverity(
    CMPIError* er,
    CMPIStatus* rc
);
```

DESCRIPTION

This function shall return the severity of the `CMPIError` object from the notifier's point of view.

The `er` argument points to the `CMPIError` object that contains the information.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The severity of the `CMPIError`.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation succeeded.
CMPI_RC_ERR_NOT_SUPPORTED	`CMPIError` is not supported.
CMPI_RC_ERR_FAILED	A generic error occurred.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIErrorFT.getProbableCause()

NAME

`CMPIErrorFT.getProbableCause()` – get the most likely cause of the error

SYNOPSIS

```
CMPIErrorProbableCause getProbableCause(
    const CMPIError* er,
    CMPIStatus* rc
);
```

DESCRIPTION

This function shall return the probable cause of the `CMPIError` object.

The `er` argument points to the `CMPIError` object that contains the information.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

An enumeration which represents the probable cause of the `CMPIError`.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation succeeded.
CMPI_RC_ERR_NOT_SUPPORTED	`CMPIError` is not supported.
CMPI_RC_ERR_FAILED	A generic error occurred.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIErrorFT.getProbableCauseDescription()

NAME

`CMPIErrorFT.getProbableCauseDescription()` – get a string describing the probable cause

SYNOPSIS

```
CMPIString* getProbableCauseDescription(
    const CMPIError* er,
    CMPIStatus* rc
);
```

DESCRIPTION

This function shall return a free-form string describing the probable cause of the error.

The `er` argument points to the `CMPIError` object that contains the information.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

This function shall return a pointer to a string which contains the description of the `CMPIError`; this string can be NULL. To see if this function failed you *must* inspect the `rc` argument.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation succeeded.
CMPI_RC_ERR_NOT_SUPPORTED	`CMPIError` is not supported.
CMPI_RC_ERR_FAILED	A generic error occurred.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIErrorFT.getRecommendedActions()

NAME

`CMPIErrorFT.getRecommendedActions()` – get an array of strings which describes actions to take

SYNOPSIS

```
CMPIArray* getRecommendedActions(
    const CMPIError* er,
    CMPIStatus* rc
);
```

DESCRIPTION

This function shall return an array of strings which describes the recommended actions that should be taken.

The `er` argument points to the `CMPIError` object that contains the information.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

This function shall return an array which contains the recommended actions; this can be NULL. To see if this function failed you *must* inspect the `rc` argument.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation succeeded.
CMPI_RC_ERR_NOT_SUPPORTED	`CMPIError` is not supported.
CMPI_RC_ERR_FAILED	A generic error occurred.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIErrorFT.getErrorSource()

NAME

`CMPIErrorFT.getErrorSource()` – get the identifying information of the entity

SYNOPSIS

```
CMPIString* getErrorSource(
    const CMPIError* er,
    CMPIStatus* rc
);
```

DESCRIPTION

This function shall return the string which describes the entity which generated the error.

The `er` argument points to the `CMPIError` object that contains the information.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

This function shall return a string which contains the data requested; this string can be NULL. To see if this function failed you *must* inspect the `rc` argument.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation succeeded.
CMPI_RC_ERR_NOT_SUPPORTED	`CMPIError` is not supported.
CMPI_RC_ERR_FAILED	A generic error occurred.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIErrorFT.getErrorSourceFormat()

NAME

`CMPIErrorFT.getErrorSourceFormat()` – get the format of the error source

SYNOPSIS

```
CMPIString getOtherErrorSourceFormat(
    const CMPIError* er,
    CMPIStatus* rc
);
```

DESCRIPTION

This function shall return a pointer to a string which describes the source of the error.

The `er` argument points to the `CMPIError` object that contains the information.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The format of the `CMPIError` error source.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation succeeded.
CMPI_RC_ERR_NOT_SUPPORTED	`CMPIError` is not supported.
CMPI_RC_ERR_FAILED	A generic error occurred.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

`setErrorSourceFormat`

CHANGE HISTORY

None.

CMPIErrorFT.getOtherErrorSourceFormat()

NAME

`CMPIErrorFT.getOtherErrorSourceFormat()` – get a string describing other values for the format

SYNOPSIS

```
CMPIString* getOtherErrorSourceFormat(
    const CMPIError* er,
    CMPIStatus* rc
);
```

DESCRIPTION

This function shall return a pointer to a string which describes the `other` source of the error. This function shall return NULL unless the `ErrorSourceFormat` is set to `other`.

The `er` argument points to the `CMPIError` object that contains the information.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The format of the `CMPIError` error source.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation succeeded.
CMPI_RC_ERR_NOT_SUPPORTED	`CMPIError` is not supported.
CMPI_RC_ERR_FAILED	A generic error occurred.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

`getErrorSourceFormat`

CHANGE HISTORY

None.

CMPIErrorFT.getCIMStatusCode()

NAME

`CMPIErrorFT.getCIMStatusCode()` – get the status code of the error

SYNOPSIS

```
CMPIrc getCIMStatusCode(
    const CMPIError* er,
    CMPIStatus* rc
);
```

DESCRIPTION

This function shall return the status code that characterizes this error object.

The `er` argument points to the `CMPIError` object that contains the information.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The underlying Status code. Please note that not all status codes apply to all situations. It is up to the MI to ensure the correct status code is set.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation succeeded.
CMPI_RC_ERR_NOT_SUPPORTED	`CMPIError` is not supported.
CMPI_RC_ERR_FAILED	A generic error occurred.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIErrorFT.getCIMStatusCodeDescription()

NAME

`CMPIErrorFT.getCIMStatusCodeDescription()` – get a string which describes the status code

SYNOPSIS

```
CMPIString* getCIMStatusCodeDescription(
    const CMPIError* er,
    CMPIStatus* rc
);
```

DESCRIPTION

This function shall return a free-form string containing a human-readable description.of the status code.

The `er` argument points to the `CMPIError` object that contains the information.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

This function shall return a pointer to a string which contains the data requested; this string can be NULL. To see if this function failed you *must* inspect the `rc` argument.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation succeeded.
CMPI_RC_ERR_NOT_SUPPORTED	`CMPIError` is not supported.
CMPI_RC_ERR_FAILED	A generic error occurred.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIErrorFT.getMessageArguments()

NAME

`CMPIErrorFT.getMessageArguments()` – get an array of strings for the dynamic content of the message

SYNOPSIS

```
CMPIArray* getMessageArguments(
    const CMPIError* er,
    CMPIStatus* rc
);
```

DESCRIPTION

This function shall return an array of `CMPIStrings`.

The `er` argument points to the `CMPIError` object that contains the information.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

This function shall return an array which contains the dynamic content of the message; this array can be NULL. To see if this function failed you *must* inspect the `rc` argument.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation succeeded.
CMPI_RC_ERR_NOT_SUPPORTED	`CMPIError` is not supported.
CMPI_RC_ERR_FAILED	A generic error occurred.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIErrorFT.setErrorType()

NAME

`CMPIErrorFT.setErrorType()` – set the primary error type

SYNOPSIS

```
CMPIStatus setErrorType(
    CMPIError* er,
    const CMPIErrorType et
);
```

DESCRIPTION

This function shall set the primary classification of the `CMPIError` type to the passed-in value.

The `er` argument specifies the `CMPIError` object to which the error type should be added.

The `et` argument specifies the primary classification of the error.

RETURN VALUE

This function shall return a `CMPIStatus` object which indicates the status of the underlying service.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation succeeded.
CMPI_RC_ERR_NOT_SUPPORTED	`CMPIError` is not supported.
CMPI_RC_ERR_INVALID_PARAMETER	One of the parameters is invalid.
CMPI_RC_ERR_FAILED	A generic error occurred.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIErrorFT.setOtherErrorType()

NAME

`CMPIErrorFT.setOtherErrorType()` – set a string which describes the error

SYNOPSIS

```
CMPIStatus setOtherErrorType(
    CMPIError* er,
    const char* ot
);
```

DESCRIPTION

This function shall set the `other` error type to the passed-in string. This string should be a free-form string describing the `ErrorType` when `ErrorType` is set to `other`.

The `er` argument specifies the `CMPIError` object to which the error type should be added.

The `ot` argument specifies a free-form string describing the `ErrorType` when `Other` is specified as the `ErrorType`. This value may be NULL.

RETURN VALUE

This function shall return a `CMPIStatus` object which indicates the status of the underlying service.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation succeeded.
CMPI_RC_ERR_NOT_SUPPORTED	`CMPIError` is not supported.
CMPI_RC_ERR_INVALID_PARAMETER	One of the parameters is invalid.
CMPI_RC_ERR_FAILED	A generic error occurred.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

`setErrorType`

CHANGE HISTORY

None.

CMPIErrorFT.setProbableCauseDescription()

NAME

`CMPIErrorFT.setProbableCauseDescription()` – set the probable cause of the error

SYNOPSIS

```
CMPIStatus setProbableCauseDescription(
    CMPIError* er,
    const char* pcd
);
```

DESCRIPTION

This function shall set the `CMPIError` probable cause. This string should be a free-form string describing the probable cause of the error.

The `er` argument specifies the `CMPIError` object to which the error type should be added.

The `pcd` argument specifies a free-form string describing the probable cause of the error. This argument can be NULL.

RETURN VALUE

This function shall return a `CMPIStatus` object which indicates the status of the underlying service.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation succeeded.
CMPI_RC_ERR_NOT_SUPPORTED	`CMPIError` is not supported.
CMPI_RC_ERR_INVALID_PARAMETER	One of the parameters is invalid.
CMPI_RC_ERR_FAILED	A generic error occurred.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIErrorFT.setRecommendedActions()

NAME

`CMPIErrorFT.setRecommendedActions()` – set the recommended actions

SYNOPSIS

```
CMPIStatus setRecommendedActions(
    CMPIError* er,
    const CMPIArray* ra
);
```

DESCRIPTION

This function shall set the `CMPIError` recommended actions.

The `er` argument specifies the `CMPIError` object to which the error type should be added.

The `ra` argument specifies an array of strings which contains free-form strings describing recommended actions to take to resolve the error. This argument may be NULL.

RETURN VALUE

This function shall return a `CMPIStatus` object which indicates the status of the underlying service.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation succeeded.
CMPI_RC_ERR_NOT_SUPPORTED	`CMPIError` is not supported.
CMPI_RC_ERR_INVALID_PARAMETER	One of the parameters is invalid.
CMPI_RC_ERR_FAILED	A generic error occurred.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIErrorFT.setErrorSource()

NAME

`CMPIErrorFT.setErrorSource()` – set the source of the error

SYNOPSIS

```
CMPIStatus setErrorSource(
    CMPIError* er,
    const char* es
);
```

DESCRIPTION

This function shall set the `CMPIError` source to the passed-in value.

The `er` argument specifies the `CMPIError` object to which the error type should be added.

The `es` argument specifies a string which identifies information of the entity generating the error (i.e., the instance). If this entity is modeled in the CIM Schema, this property contains the path of the instance encoded as a string parameter. If not modeled, the property contains some identifying string that names the entity that generated the error. The path or identifying string is formatted per the `ErrorSourceFormat` property. This argument can be NULL.

RETURN VALUE

This function shall return a `CMPIStatus` object which indicates the status of the underlying service.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation succeeded.
CMPI_RC_ERR_NOT_SUPPORTED	`CMPIError` is not supported.
CMPI_RC_ERR_INVALID_PARAMETER	One of the parameters is invalid.
CMPI_RC_ERR_FAILED	A generic error occurred.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIErrorFT.setErrorSourceFormat()

NAME

`CMPIErrorFT.setErrorSourceFormat()` – set the format of the error

SYNOPSIS

```
CMPIStatus setErrorSourceFormat(
    CMPIError* er,
    const CMPIErrorSrcFormat esf
);
```

DESCRIPTION

This function shall set the `CMPIError` type to the passed-in value.

The `er` argument specifies the `CMPIError` object to which the error type should be added.

The `esf` argument specifies the error source format. The format of the `ErrorSource` property is interpretable based on the value of this parameter.

RETURN VALUE

This function shall return a `CMPIStatus` object which indicates the status of the underlying service.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation succeeded.
CMPI_RC_ERR_NOT_SUPPORTED	`CMPIError` is not supported.
CMPI_RC_ERR_INVALID_PARAMETER	One of the parameters is invalid.
CMPI_RC_ERR_FAILED	A generic error occurred.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIErrorFT.setOtherErrorSourceFormat()

NAME

`CMPIErrorFT.setOtherErrorSourceFormat()` – set the value of the `other` source format

SYNOPSIS

```
CMPIStatus setOtherErrorSourceFormat(
    CMPIError* er,
    const char* oef
);
```

DESCRIPTION

This function shall set the `CMPIError` type to the passed-in value; this function shall fail unless the `CMPIError` error source format is set to `other`.

The `er` argument specifies the `CMPIError` object to which the error type should be added.

The `oef` argument specifies a string defining `Other` values for `ErrorSourceFormat`. This value *must* be set to a non-NULL value when `ErrorSourceFormat` is set to a value of `Other`. For all other values of `ErrorSourceFormat`, the value of this string *must* be set to NULL.

RETURN VALUE

This function shall return a `CMPIStatus` object which indicates the status of the underlying service.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation succeeded.
CMPI_RC_ERR_NOT_SUPPORTED	`CMPIError` is not supported.
CMPI_RC_ERR_INVALID_PARAMETER	One of the parameters is invalid.
CMPI_RC_ERR_FAILED	A generic error occurred.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

`setErrorSourceFormat`

CHANGE HISTORY

None.

CMPIErrorFT.setCIMStatusCodeDescription()

NAME

`CMPIErrorFT.setCIMStatusCodeDescription()` – set the status code description

SYNOPSIS

```
CMPIStatus setCIMStatusCodeDescription(
    CMPIError* er,
    const char* scd
);
```

DESCRIPTION

This function shall set the `CMPIError` type to the passed-in value.

The `er` argument specifies the `CMPIError` object to which the error type should be added.

The `scd` argument specifies a free-form string containing a human-readable description of `CIMStatusCode`. This description *may* extend, but *must* be consistent with, the definition of Status Code.

RETURN VALUE

This function shall return a `CMPIStatus` object which indicates the status of the underlying service.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation succeeded.
CMPI_RC_ERR_NOT_SUPPORTED	`CMPIError` is not supported.
CMPI_RC_ERR_INVALID_PARAMETER	One of the parameters is invalid.
CMPI_RC_ERR_FAILED	A generic error occurred.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIErrorFT.setMessageArguments()

NAME

`CMPIErrorFT.setMessageArguments()` – set an array of strings for the dynamic content of the message

SYNOPSIS

```
CMPIStatus setMessageArguments(
    CMPIError* er,
    CMPIArray* values
);
```

DESCRIPTION

This function shall set the `CMPIErrors` message arguments to the passed-in values. This is an array supplied by the MI and is not directly used by the MB.

The `er` argument specifies the `CMPIError` object to which the error type should be added.

The `values` argument specifies an array containing the dynamic content of the message.

RETURN VALUE

This function shall return a `CMPIStatus` object which indicates the status of the underlying service.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation succeeded.
CMPI_RC_ERR_NOT_SUPPORTED	`CMPIError` is not supported.
CMPI_RC_ERR_INVALID_PARAMETER	One of the parameters is invalid.
CMPI_RC_ERR_FAILED	A generic error occurred.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

7 Qualifier Support

Qualifier support is an optional feature of CMPI.

Availability of this support is indicated by the `CMPI_MB_BasicQualifierSupport` flag in `CMPIBrokerFT.brokerCapabilities`.

Qualifier support is a subset of full schema support. It entails read-only access to CIM type qualifiers of class definitions and its components. Qualifier support is an extension to the `CMPIObjectPath` encapsulated object. The model path portion of the `CMPIObjectPath` is used to identify the object.

Qualifier support is made available via the `CMPIObjectPathFT` function table.

Qualifier support is provided by the following functions.

CMPIObjectPathFT.getClassQualifier()

NAME

`CMPIObjectPathFT.getClassQualifier()` – get the class qualifier value

SYNOPSIS

```
CMPIData CMPIObjectPathFT.getClassQualifier(
    const CMPIObjectPath* op,
    const char* qName,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIObjectPathFT.getClassQualifier()` function shall return the class qualifier value of an `ObjectPath`.

The `op` argument points to a `CMPIObjectPath` structure.

The `qName` argument is a string containing the qualifier name whose value is to be returned.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIObjectPathFT.getClassQualifier()` function shall return a `CMPIData` structure containing the qualifier value. In case of error, `CMPIData.state` shall be set to `false`.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_NOT_SUPPORTED	Qualifier operations not supported.
CMPI_RC_ERR_INVALID_HANDLE	The `op` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIObjectPathFT.getMethodQualifier()

NAME

`CMPIObjectPathFT.getMethodQualifier()` – get the method qualifier value

SYNOPSIS

```
CMPIData CMPIObjectPathFT.getMethodQualifier(
    const CMPIObjectPath* op,
    const char* mName,
    const char* qName,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIObjectPathFT.getMethodQualifier()` function shall return the qualifier value of the specified method from the class represented by the `ObjectPath`.

The `op` argument points to a `CMPIObjectPath` structure.

The `mName` argument is a string containing the method name.

The `qName` argument is a string containing the qualifier name whose value is to be returned.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIObjectPathFT.getMethodQualifier()` function shall return a `CMPIData` structure containing the qualifier value. In case of error, `CMPIData.state` shall be set to `false`.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_NOT_SUPPORTED	Qualifier operations not supported.
CMPI_RC_ERR_INVALID_HANDLE	The `op` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIObjectPathFT.getParameterQualifier()

NAME

`CMPIObjectPathFT.getParameterQualifier()` – get the method parameter qualifier value

SYNOPSIS

```
CMPIData CMPIObjectPathFT.getParameterQualifier(
    const CMPIObjectPath* op,
    const char* mName,
    const char* pName,
    const char* qName,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIObjectPathFT.getParamaterQualifier()` function shall return the qualifier value of the specified method from the class represented by the `ObjectPath`.

The `op` argument points to a `CMPIObjectPath` structure.

The `mName` argument is a string containing the method name.

The `pName` argument is a string containing the parameter name.

The `qName` argument is a string containing the qualifier name whose value is to be returned.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIObjectPathFT.getParameterQualifier()` function shall return a `CMPIData` structure containing the qualifier value. In case of error, `CMPIData.state` shall be set to `false`.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_NOT_SUPPORTED	Qualifier operations not supported.
CMPI_RC_ERR_INVALID_HANDLE	The `op` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIObjectPathFT.getPropertyQualifier()

NAME

`CMPIObjectPathFT.getPropertyQualifier()` – get the property qualifier

SYNOPSIS

```
CMPIData CMPIObjectPathFT.getPropertyQualifier(
    const CMPIObjectPath* op,
    const char* pName,
    const char* qName,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIObjectPathFT.getPropertyQualifier()` function shall return the qualifier value of the specified method from the class represented by the `ObjectPath`.

The `op` argument points to a `CMPIObjectPath` structure.

The `pName` argument is a string containing the property name.

The `qName` argument is a string containing the qualifier name whose value is to be returned.

The `rc` output argument, if not NULL, is used to return a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIObjectPathFT.getPropertyQualifier()` function shall return a `CMPIData` structure containing the qualifier value. In case of error, `CMPIData.state` shall be set to `false`.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_NOT_SUPPORTED	Qualifier operations not supported.
CMPI_RC_ERR_INVALID_HANDLE	The `op` handle is invalid.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

8 CMPIBrokerFT.brokerCapabilities MB Services

Management Broker (MB) services are part of CMPI support. In general, the services correspond with most of the services either available via a `CIMClient` interface or via various `CIMOMProviderHandle` implementations.

8.1 MB Capabilities

MBs may not support all MB services listed in this chapter. CMPI uses a scheme to define capabilities similar to the DMTF's Functional Grouping scheme. Management Instrumentation (MI) can query the MB capabilities by inspecting the `brokerCapabilities` field of the `CMPIBrokerFT` structure.

Capability	**Dependency**	**CMPIBrokerFT Functions**

Basic Read	None	`GetInstance`
		`EnumerateInstances`
		`EnumerateInstanceNames`
		`GetProperty`

Basic Write	Basic Read	`SetProperty`

Instance Manipulation	Basic Write	`CreateInstance`
		`ModifyInstance`
		`DeleteInstance`
		`InvokeMethod`

Association Traversal	Basic Read	`Associators`
		`AssociatorNames`
		`References`
		`ReferenceNames`

Query Execution	Basic Read	`ExecQuery`
Indications	Basic Read	`DeliverIndication`

Capability	**Dependency**	**Encapsulated Types**

Query Normalization	Query Execution	`CMPISelectExp`
		`CMPISelectCond`
		`CMPISubCond`
		`CMPIPredicates`

Capability	**Dependency**	**CMPIObjectPathFT Functions**

Basic Qualifier Support	Instance Manipulation	`getClassQualifier`
		`getMethodQualifier`
		`getParameterQualifier`
		`getPropertyQualifier`

Capability	Dependency	CMPIErrorFT Functions

Extended Error Support	None	`Clone`
		`release`
		`getErrorType`
		`getOtherErrorType`
		`getOwningEntity`
		`getMessageID`
		`getMessage`
		`getPerceivedSeverity`
		`getProbablyCause`
		`getProbableCauseDescription`
		`getRecommendedActions`
		`getErrorSource`
		`getErrorSourceFormat`
		`getOtherErrorSourceFormat`
		`getCIMStatusCode`
		`getCIMStatusCodeDescription`
		`setErrorType`
		`setOtherErrorType`
		`setProbableCauseDescription`
		`setRecommendedActions`
		`setErrorSource`
		`setErrorSourceFormat`
		`setOtherErrorSourceFormat`
		`getCIMStatusCode`
		`getCIMStatusCodeDescription`

Capability	Dependency	CMPIBrokerExtFT Functions

OS Encapsulation	None	`newThread`
		`joinThread`
		`exitThread`
		`cancelThread`
		`threadSleep`
		`threadOnce`
		`createthreadKey`
		`destroyThreadKey`
		`getThreadSpecific`
		`setThreadSpecific`
		`newMutex`
		`destroyMutex`
		`lockMutex`
		`unlockMutex`
		`newCondition`
		`destroyCondition`
		`condWait`
		`timedCondWait`
		`signalCondition`

Capability	Dependency	CMPIBrokerMemFT Functions
Memory Enhancements	Noine	mark release cmpiMalloc cmpiCalloc cmpiRealloc cmpiStrDup cmpiFree freeInstance freeObjectPath freeArgs freeString freeArray freeDateTime freeSelectExp

MIs can query the MB capabilities and decide to refuse to operate in a less-than-needed environment, or curtail its functionality. This can be done by the MI factory by returning CMPI_RC_ERR_NOT_SUPPORTED.

The following are valid values for `CMPIBrokerFT.brokerCapabilities`:

```
#define CMPI_MB_BasicRead                    0x00000001
#define CMPI_MB_BasicWrite                   0x00000003
#define CMPI_MB_InstanceManipulation         0x00000007
#define CMPI_MB_AssociationTraversal         0x00000009
#define CMPI_MB_QueryExecution               0x00000011
#define CMPI_MB_QueryNormalization           0x00000031
#define CMPI_MB_BasicQualifierSupport        0x00000047
#define CMPI_MB_Indications                  0x00000081
#define CMPI_MB_OSEncapsulationSupport       0x00000100
#define CMPI_MB_Supports MemEnhancements     0x00004000
#define CMPI_MB_Supports_Extended_Error      0x00008000
```

MB support is made available via the `CMPIBrokerFT` function table:

```
typedef struct _CMPIBroker {
    const void* hdl;
    const CMPIBrokerFT* bft;
    const CMPIBrokerEncFT* eft;
    const CMPIBrokerExtFT* xft;
    const CMPIBrokerMemFT* mft;
} CMPIBroker;
```

Notice that `CMPIBrokerMemFT` functions are optional; they are only available when the `CMPI_MB_Supports_MemEnhancements` flag in `CMPIBrokerFT.brokerCapabilities` is set. Otherwise, this pointer shall be initialized to zero.

```
typedef struct _CMPIBrokerFT {
    const unsigned int brokerCapabilities;
    const unsigned int brokerVersion;
    const char* brokerName;
    CMPIContext (*prepareAttach)
        (const CMPIBroker*,const CMPIContext*);
```

```
CMPIStatus (*attachThread)
    (const CMPIBroker*,const CMPIContext*);
CMPIStatus (*detachThread)
    (const CMPIBroker*,const CMPIContext*);
CMPIStatus (*deliverIndication)
    (const CMPIBroker*,const CMPIContext*,
     const char*,const CMPIInstance*);
CMPIEnumeration* (*enumerateInstanceNames)
    (const CMPIBroker*,const CMPIContext*,
     const CMPIObjectPath*,CMPIStatus*);
CMPIInstance* (*getInstance)
    (const CMPIBroker*,const CMPIContext*,
     const CMPIObjectPath*,const char**,CMPIStatus*);
CMPIObjectPath* (*createInstance)
    (const CMPIBroker*,const CMPIContext*,
     const CMPIOBjectPath*,const CMPIInstance*,
     CMPIStatus*);
CMPIStatus (*modifyInstance)
    (const CMPIBroker*,const CMPIContext*,
     const CMPIObjectPath*,CMPIInstance*,const char**);
CMPIStatus (*deleteInstance)
    (const CMPIBroker*,const CMPIContext*,
     const CMPIObjectPath*);
CMPIEnumeration* (*execQuery)
    (const CMPIBroker*,const CMPIContext*,
     const CMPIObjectPath*,const char*,
     const char*,CMPIStatus*);
CMPIEnumeration* (*enumerateInstances)
    (const CMPIBroker*,const CMPIContext*,
     const CMPIObjectPath*,const char**,CMPIStatus*);
CMPIEnumeration* (*associators)
    (const CMPIBroker*,const CMPIContext*,
     const CMPIObjectPath*,const char*,const char*,
     const char*,const char*,const char**,
     CMPIStatus*);
CMPIEnumeration* (*associatorNames)
    (const CMPIBroker*,const CMPIContext*,
     const CMPIObjectPath*,const char*,const char*,
     const char*,const char*,CMPIStatus*);
CMPIEnumeration* (*references)
    (const CMPIBroker*,const CMPIContext*,
     const CMPIObjectPath*,const char*,const char*,
     const char**, CMPIStatus*);
CMPIEnumeration* (*referenceNames)
    (const CMPIBroker*,const CMPIContext*,
     const CMPIObjectPath*,const char*,const char*,
     CMPIStatus*);
CMPIData (*invokeMethod)
    (const CMPIBroker*,const CMPIContext*,
     const CMPIObjectPath*,const char*,const CMPIArgs*,
     CMPIArgs*, CMPIStatus*);
CMPIStatus (*setProperty)
    (const CMPIBroker*,const CMPIContext*,
```

```
          const CMPIObjectPath*,const char*,
          const CMPIValue*,CMPIType);
     CMPIData (*getProperty)
         (const CMPIBroker*,const CMPIContext*,
          const CMPIObjectPath*,const char*,CMPIStatus*);
} CMPIBrokerFT;
```

8.2 Indications Services

The Indications services are provided by the following function.

CMPIBrokerFT.deliverIndication()

NAME

`CMPIBrokerFT.deliverIndication()` – request delivery of an Indication

SYNOPSIS

```
CMPIStatus CMPIBrokerFT.deliverIndication(
    const CMPIBroker* mb,
    const CMPIContext* ctx,
    const char* ns,
    const CMPIInstance* ind
);
```

DESCRIPTION

The `CMPIBrokerFT.deliverIndication()` requests the delivery of an Indication. The CIMOM will locate pertinent subscribers and notify them about the event.

The `mb` argument points to a `CMPIBroker` structure.

The `ctx` argument points to the context object.

The `ns` argument is a pointer to the Namespace.

The `ind` argument is a pointer to a `CMPIInstance` object containing the Indication.

RETURN VALUE

The `CMPIBrokerFT.deliverIndication()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_NOT_SUPPORTED	Operations not supported.
CMPI_RC_ERR_INVALID_HANDLE	Invalid encapsulated type handle.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

8.3 Basic Read Services

Basic Read is supported by the following functions.

CMPIBrokerFT.getInstance()

NAME

`CMPIBrokerFT.getInstance()` – get an Instance using an `ObjectPath` as a reference

SYNOPSIS

```
CMPIInstance* CMPIBrokerFT.getInstance(
    const CMPIBroker* mb,
    const CMPIContext* ctx,
    const CMPIObjectPath* op,
    const char** properties,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIBrokerFT.getInstance()` function shall retrieve an Instance using an `ObjectPath` as a reference.

The `mb` argument points to a `CMPIBroker` structure.

The `ctx` argument points to the context object.

The `op` argument points to the source `ObjectPath` containing namespace, classname, and key components.

The `properties` argument, if not NULL, is an array of elements defining one or more Property names. Each returned Object must not include elements for any Properties missing from this list. The end of the list is indicated by a NULL pointer.

The `rc` argument points to a `CMPIStatus` structure used to return the service return code.

RETURN VALUE

The `CMPIBrokerFT.getInstance()` returns a pointer to a `CMPIInstance` structure.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_ERR_NOT_SUPPORTED	Operation not supported by this MI.
CMPI_RC_ERR_ACCESS_DENIED	Not authorized.
CMPI_RC_ERR_NOT_FOUND	Instance not found.
CMPI_RC_ERR_INVALID_HANDLE	Invalid encapsulated type handle.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerFT.enumerateInstances()

NAME

`CMPIBrokerFT.enumerateInstances()` – enumerate Instances of the class (and subclasses) defined by an `ObjectPath`

SYNOPSIS

```
CMPIEnumeration* CMPIBrokerFT.enumerateInstances(
    const CMPIBroker* mb,
    const CMPIContext* ctx,
    const CMPIObjectPath* op,
    const char** properties;
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIBrokerFT.enumerateInstances()` function shall enumerate the Instance Names of the class (and subclasses) defined by the specified `ObjectPath`. Instance structures can be controlled using the `CMPIInvocationFlags` entry in `ctx`.

The `mb` argument points to a `CMPIBroker` structure.

The `ctx` argument points to the context object.

The `op` argument points to the source `ObjectPath` containing namespace and classname elements.

The `properties` argument, if not NULL, is an array of elements defining one or more Property names. Each returned Object must not include elements for any Properties missing from this list.

The `rc` argument points to a `CMPIStatus` structure used to return the service return code.

RETURN VALUE

The `CMPIBrokerFT.enumerateInstances()` function shall return a pointer to a `CMPIEnumeration` structure containing the `ObjectPaths`.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_ERR_NOT_SUPPORTED	Operation not supported by this MI.
CMPI_RC_ERR_ACCESS_DENIED	Not authorized.
CMPI_RC_ERR_NOT_FOUND	Instance not found.
CMPI_RC_ERR_INVALID_HANDLE	Invalid encapsulated type handle.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerFT.enumerateInstanceNames()

NAME

`CMPIBrokerFT.enumerateInstanceNames()` – enumerate the Instance Names of the class (and subclasses) defined by an `ObjectPath`

SYNOPSIS

```
CMPIEnumeration* CMPIBrokerFT.enumerateInstanceNames(
    const CMPIBroker* mb,
    const CMPIContext* ctx,
    const CMPIObjectPath* op,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIBrokerFT.enumerateInstanceNames()` function shall enumerate the Instance Names of the class (and subclasses) defined by the specified `ObjectPath`. Instance structures can be controlled using the `CMPIInvocationFlags` entry in `ctx`.

The `mb` argument points to a `CMPIBroker` structure.

The `ctx` argument points to the context object.

The `op` argument points to the source `ObjectPath` containing namespace and classname elements.

The `rc` argument points to a `CMPIStatus` structure used to return the service return code.

RETURN VALUE

The `CMPIBrokerFT.enumerateInstanceNames()` function shall return a pointer to a `CMPIEnumeration` structure containing the `ObjectPaths`.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_ERR_NOT_SUPPORTED	Operation not supported by this MI.
CMPI_RC_ERR_ACCESS_DENIED	Not authorized.
CMPI_RC_ERR_NOT_FOUND	Instance not found.
CMPI_RC_ERR_INVALID_HANDLE	Invalid encapsulated type handle.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerFT.getProperty()

NAME

`CMPIBrokerFT.getProperty()` – get the named property value of an Instance defined by an `ObjectPath`

SYNOPSIS

```
CMPIData CMPIBrokerFT.getProperty(
    const CMPIBroker* mb,
    const CMPIContext* ctx,
    const CMPIObjectPath* op,
    const char* name,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIBrokerFT.getProperty()` function shall get the named property value of an Instance defined by an `ObjectPath`.

The `mb` argument points to a `CMPIBroker` object.

The `ctx` argument points to the context object.

The `op` argument points to the source `ObjectPath` containing namespace, classname, and key elements.

The `name` argument points to a string containing the property name.

The `rc` argument points to a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIBrokerFT.getProperty()` function shall return a `CMPIData` structure containing the requested property value.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_ERR_NOT_SUPPORTED	Operation not supported by this MI.
CMPI_RC_ERR_ACCESS_DENIED	Not authorized.
CMPI_RC_ERR_NOT_FOUND	Instance not found.
CMPI_RC_ERR_NO_SUCH_PROPERTY	Property not found.
CMPI_RC_ERR_INVALID_HANDLE	Invalid encapsulated type handle.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

8.4 Basic Write Service

Basic Write is supported by the following function.

CMPIBrokerFT.setProperty()

NAME

`CMPIBrokerFT.setProperty()` – set the named property value of an Instance defined by an `ObjectPath`

SYNOPSIS

```
CMPIStatus CMPIBrokerFT.setProperty(
    const CMPIBroker* mb,
    const CMPIContext* context,
    const CMPIObjectPath* op,
    const char* name,
    const CMPIValue* value,
    CMPIType type
);
```

DESCRIPTION

The `CMPIBrokerFT.setProperty()` function shall set the named property value of an Instance defined by an `ObjectPath`.

The `mb` argument points to a `CMPIBroker` structure.

The `ctx` argument points to the context object.

The `op` argument points to the source `ObjectPath` containing namespace, classname, and key components.

The `name` argument points to a string containing the property name.

The `value` argument points to a `CMPIValue` structure containing the value to be assigned to the property.

The `type` argument points to a `CMPIType` structure which defines the type of `value`.

RETURN VALUE

The `CMPIBrokerFT.setProperty()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_ERR_NOT_SUPPORTED	Operation not supported by this MI.
CMPI_RC_ERR_ACCESS_DENIED	Not authorized.
CMPI_RC_ERR_NOT_FOUND	Instance not found.
CMPI_RC_ERR_NO_SUCH_PROPERTY	Property not found.
CMPI_RC_ERR_TYPE_MISMATCH	Value types incompatible.
CMPI_RC_ERR_INVALID_HANDLE	Invalid encapsulated type handle.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

8.5 Instance Manipulation Services

Instance Manipulation is supported by the following functions.

CMPIBrokerFT.createInstance()

NAME

`CMPIBrokerFT.createInstance()` – create an Instance using a specified `ObjectPath` as a reference

SYNOPSIS

```
CMPIObjectPath* CMPIBrokerFT.createInstance(
    const CMPIBroker* mb,
    const CMPIContext* ctx,
    const CMPIOjectPath* op,
    const CMPIInstance* inst,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIBrokerFT.createInstance()` function shall create an Instance using a specified `ObjectPath` as a reference.

The `mb` argument points to a `CMPIBroker` structure.

The `ctx` argument points to the context object.

The `op` argument points to the source `ObjectPath` containing namespace, classname, and key components.

The `inst` argument points to a `CMPIInstance` structure.

The `rc` argument points to a `CMPIStatus` structure used to return the service return code.

RETURN VALUE

The `CMPIBrokerFT.createInstance()` function shall return a pointer to the assigned Instance reference.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_ERR_NOT_SUPPORTED	Operation not supported by this MI.
CMPI_RC_ERR_ALREADY_EXISTS	Instance already exists.
CMPI_RC_ERR_INVALID_HANDLE	Invalid encapsulated type handle.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerFT.modifyInstance()

NAME

`CMPIBrokerFT.modifyInstance()` – replace an existing Instance using an `ObjectPath` as reference

SYNOPSIS

```
CMPIStatus CMPIBrokerFT.modifyInstance(
    const CMPIBroker* mb,
    const CMPIContext* ctx,
    const CMPIObjectPath* op,
    const CMPIInstance* inst,
    const char** propertyList
);
```

DESCRIPTION

The `CMPIBrokerFT.modifyInstance()` function shall replace an existing Instance using an `ObjectPath` as a reference.

The `mb` argument points to a `CMPIBroker` structure.

The `ctx` argument points to the context object.

The `op` argument points to the source `ObjectPath` containing namespace, classname, and key components.

The `inst` argument points to a `CMPIInstance` structure containing a complete instance.

The `propertyList` argument, if not NULL, defines the list of property names to be modified. If NULL, all properties will be modified. The end of the list is signalled by a NULL character pointer.

RETURN VALUE

The `CMPIBrokerFT.modifyInstance()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_ERR_NOT_SUPPORTED	Operation not supported by this MI.
CMPI_RC_ERR_ACCESS_DENIED	Not authorized.
CMPI_RC_ERR_NOT_FOUND	Instance not found.
CMPI_RC_ERR_INVALID_HANDLE	Invalid encapsulated type handle.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerFT.deleteInstance()

NAME

`CMPIBrokerFT.deleteInstance()` – delete an existing Instance using a specified `ObjectPath` as a reference

SYNOPSIS

```
CMPIStatus CMPIBrokerFT.deleteInstance(
    const CMPIBroker* mb,
    const CMPIContext* ctx,
    const CMPIObjectPath* op
);
```

DESCRIPTION

The `CMPIBrokerFT.deleteInstance()` function shall delete an existing Instance using a specified `ObjectPath` as a reference.

The `mb` argument points to a `CMPIBroker` structure.

The `ctx` argument points to the context object.

The `op` argument points to the source `ObjectPath` containing namespace, classname, and key components.

RETURN VALUE

The `CMPIBrokerFT.deleteInstance()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_ERR_NOT_SUPPORTED	Operation not supported by this MI.
CMPI_RC_ERR_ACCESS_DENIED	Not authorized.
CMPI_RC_ERR_NOT_FOUND	Instance not found.
CMPI_RC_ERR_INVALID_HANDLE	Invalid encapsulated type handle.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerFT.invokeMethod()

NAME

`CMPIBrokerFT.invokeMethod()` – invoke a named, extrinsic method of an Instance defined by an `ObjectPath`

SYNOPSIS

```
CMPIData CMPIBrokerFT.invokeMethod(
    const CMPIBroker* mb,
    const CMPIContext* ctx,
    const CMPIObjectPath* op,
    const char* method,
    const CMPIArgs* in,
    CMPIArgs* out,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIBrokerFT.invokeMethod()` function shall invoke a named, extrinsic method of an instance defined by a specified `ObjectPath`.

The `mb` argument points to a `CMPIBroker` object.

The `ctx` argument points to the context object.

The `op` argument points to the source `ObjectPath` containing namespace, classname, and key elements.

The `method` argument points to a string containing the method name.

The `in` argument points to a `CMPIArgs` structure containing the input parameters.

The `out` argument points to a `CMPIArgs` structure containing the output parameters.

The `rc` argument points to a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIBrokerFT.invokeMethod()` function shall return a `CMPIData` structure containing the method return value.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_ERR_NOT_SUPPORTED	Operation not supported by this MI.
CMPI_RC_ERR_ACCESS_DENIED	Not authorized.
CMPI_RC_ERR_NOT_FOUND	Instance not found.
CMPI_RC_ERR_METHOD_NOT_AVAILABLE	Method not available.
CMPI_RC_ERR_METHOD_NOT_FOUND	Method not found.
CMPI_RC_ERR_INVALID_HANDLE	Invalid encapsulated type handle.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

8.6 Asscociation Traversal Services

Asscociation Tarversal is supported by the following functions.

CMPIBrokerFT.associators()

NAME

`CMPIBrokerFT.associators()` – enumerate Instances associated with an Instance

SYNOPSIS

```
CMPIEnumeration* CMPIBrokerFT.associators(
    const CMPIBroker* mb,
    const CMPIContext* ctx,
    const CMPIObjectPath* op,
    const char* assocClass,
    const char* resultClass,
    const char* role,
    const char* resultRole,
    const char** properties,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIBrokerFT.associators()` function shall enumerate Instances associated with an Instance.

The `mb` argument points to a `CMPIBroker` structure.

The `ctx` argument points to the context object.

The `op` argument points to the source `ObjectPath` containing namespace, classname, and key components.

The `assocClass` argument, if not NULL, shall be a valid Association Class name. It acts as a filter on the returned set of Objects by mandating that each returned Object must be associated to the source Object via an Instance of this Class or one of its subclasses.

The `resultclass` argument, if not NULL, shall be a valid Class name. It acts as a filter on the returned set of Objects by mandating that each returned Object must be either an Instance of this Class (or one of its subclasses).

The `role` argument, if not NULL, shall be a valid Property name. It acts as a filter on the returned set of Objects by mandating that each returned Object must be associated to the source Object via an Association in which the source Object plays the specified role (i.e., the name of the Property in the Association Class that refers to the source Object must match the value of this parameter).

The `resultrole` argument, if not NULL, shall be a valid Property name. It acts as a filter on the returned set of Objects by mandating that each returned Object must be associated to the source Object via an Association in which the returned Object plays the specified role (i.e., the name of the Property in the Association Class that refers to the returned Object must match the value of this parameter).

The `properties` argument, if not NULL, is an array of elements defining one or more Property names. Each returned Object must not include elements for any Properties missing from this list. The end of the list is indicated by a NULL pointer.

The `rc` argument points to a `CMPIStatus` structure used to return the service return code.

RETURN VALUE

The `CMPIBrokerFT.associators()` function shall return a pointer to a `CMPIEnumeration` structure containing the enumeration of the Instances.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_ERR_NOT_SUPPORTED	Operation not supported by this MI.
CMPI_RC_ERR_ACCESS_DENIED	Not authorized.
CMPI_RC_ERR_NOT_FOUND	Instance not found.
CMPI_RC_ERR_INVALID_HANDLE	Invalid encapsulated type handle.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerFT.associatorNames()

NAME

`CMPIBrokerFT.associatorNames()` – enumerate `ObjectPaths` of Instances associated with an Instance

SYNOPSIS

```
CMPIEnumeration* CMPIBrokerFT.associatorNames(
    const CMPIBroker* mb
    const CMPIContext* ctx,
    const CMPIObjectPath* op,
    const char* assocClass,
    const char* resultClass,
    const char* role,
    const char* resultRole,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIBrokerFT.associatorNames()` function shall enumerate `ObjectPaths` of Instances associated with an Instance.

The `mb` argument points to a `CMPIBroker` structure.

The `ctx` argument points to the context object.

The `op` argument points to the source `ObjectPath` containing namespace, classname, and key components.

The `assocClass` argument, if not NULL, shall be a valid Association Class name. It acts as a filter on the returned set of Objects by mandating that each returned Object must be associated to the source Object via an Instance of this Class or one of its subclasses.

The `resultclass` argument, if not NULL, shall be a valid Class name. It acts as a filter on the returned set of Objects by mandating that each returned Object must be either an Instance of this Class (or one of its subclasses).

The `role` argument, if not NULL, shall be a valid Property name. It acts as a filter on the returned set of Objects by mandating that each returned Object must be associated to the source Object via an Association in which the source Object plays the specified role (i.e., the name of the Property in the Association Class that refers to the source Object must match the value of this parameter).

The `resultrole` argument, if not NULL, shall be a valid Property name. It acts as a filter on the returned set of Objects by mandating that each returned Object must be associated to the source Object via an Association in which the returned Object plays the specified role (i.e., the name of the Property in the Association Class that refers to the returned Object must match the value of this parameter).

RETURN VALUE

The `CMPIBrokerFT.associatorNames()` function shall return a pointer to a `CMPIEnumeration` structure containing the enumeration of the `ObjectPaths`.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.

CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_ERR_NOT_SUPPORTED	Operation not supported by this MI.
CMPI_RC_ERR_ACCESS_DENIED	Not authorized.
CMPI_RC_ERR_NOT_FOUND	Instance not found.
CMPI_RC_ERR_INVALID_HANDLE	Invalid encapsulated type handle.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerFT.references()

NAME

`CMPIBrokerFT.references()` – enumerate the association Instances that refer to an Instance defined by an `ObjectPath`

SYNOPSIS

```
CMPIEnumeration* CMPIBrokerFT.references(
    const CMPIBroker* mb,
    const CMPIContext* ctx,
    const CMPIObjectPath* op,
    const char* resultClass,
    const char* role,
    const char** properties,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIBrokerFT.references()` function shall enumerate the association Instances that refer to an Instance defined by an `ObjectPath`.

The `mb` argument points to a `CMPIBroker` structure.

The `ctx` argument points to the context object.

The `op` argument points to the source `ObjectPath` containing namespace, classname, and key components.

The `resultclass` argument, if not NULL, shall be a valid Class name. It acts as a filter on the returned set of Objects by mandating that each returned Object must be either an Instance of this Class (or one of its subclasses).

The `role` argument, if not NULL, shall be a valid Property name. It acts as a filter on the returned set of Objects by mandating that each returned Object must be associated to the source Object via an Association in which the source Object plays the specified role (i.e., the name of the Property in the Association Class that refers to the source Object must match the value of this parameter).

The `properties` argument, if not NULL, is an array of elements defining one or more Property names. Each returned Object must not include elements for any Properties missing from this list. The end of the list is indicated by a NULL pointer.

The `rc` argument points to a `CMPIStatus` structure used to return the service return code.

RETURN VALUE

The `CMPIBrokerFT.references()` function shall return a pointer to a `CMPIEnumeration` structure containing the enumeration of the `ObjectPaths`.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_ERR_NOT_SUPPORTED	Operation not supported by this MI.
CMPI_RC_ERR_ACCESS_DENIED	Not authorized.

CMPI_RC_ERR_NOT_FOUND	Instance not found.
CMPI_RC_ERR_INVALID_HANDLE	Invalid encapsulated type handle.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerFT.referenceNames()

NAME

`CMPIBrokerFT.referenceNames()` – enumerate the `ObjectPaths` of association Instances that refer to an Instance defined by an `ObjectPath`

SYNOPSIS

```
CMPIEnumeration* CMPIBrokerFT.referenceNames(
    const CMPIBroker* mb
    const CMPIContext* ctx,
    const CMPIObjectPath* op,
    const char* resultClass,
    const char* role,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIBrokerFT.referenceNames()` function shall enumerate the `ObjectPaths` of association Instances that refer to an Instance defined by an `ObjectPath`.

The `mb` argument points to a `CMPIBroker` structure.

The `ctx` argument points to the context object.

The `op` argument points to the source `ObjectPath` containing namespace, classname, and key components.

The `resultclass` argument, if not NULL, shall be a valid Class name. It acts as a filter on the returned set of Objects by mandating that each returned Object must be either an Instance of this Class (or one of its subclasses).

The `role` argument, if not NULL, shall be a valid Property name. It acts as a filter on the returned set of Objects by mandating that each returned Object must be associated to the source Object via an Association in which the source Object plays the specified role (i.e., the name of the Property in the Association Class that refers to the source Object must match the value of this parameter).

The `rc` argument points to a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIBrokerFT.referenceNames()` function shall return a pointer to a `CMPIEnumeration` structure containing the enumeration of the `ObjectPaths`.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_ERR_NOT_SUPPORTED	Operation not supported by this MI.
CMPI_RC_ERR_ACCESS_DENIED	Not authorized.
CMPI_RC_ERR_NOT_FOUND	Instance not found.
CMPI_RC_ERR_INVALID_HANDLE	Invalid encapsulated type handle.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

8.7 Query Execution Service

Query Execution is supported by the following function.

CMPIBrokerEncFT.execQuery()

NAME

`CMPIBrokerEncFT.execQuery()` – query the enumeration of Instances of the class (and subclasses) defined by an `ObjectPath` using a query expression

SYNOPSIS

```
CMPIEnumeration* CMPIBrokerEncFT.execQuery(
    const CMPIBroker* mb,
    const CMPIContext* ctx,
    const CMPIObjectPath* op,
    const char* query,
    const char* lang,
    CMPIStatus* rc
);
```

DESCRIPTION

The `CMPIBrokerEncFT.execQuery()` function shall return a new `CMPISelectExp` object.

The `mb` argument points to a `CMPIBroker` object.

The `ctx` argument points to the context object.

The `op` argument points to the source `ObjectPath` containing namespace and classname elements.

The `query` argument is a pointer to a string containing the select expression.

The `lang` argument is a pointer to a string containing the query language.

The `rc` argument points to a `CMPIStatus` structure containing the service return status.

RETURN VALUE

The `CMPIBrokerEncFT.execQuery()` function shall return a pointer to a `CMPIEnumeration` object containing the Instances.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.
CMPI_RC_ERR_NOT_SUPPORTED	Operation not supported by this MI.
CMPI_RC_ERR_ACCESS_DENIED	Not authorized.
CMPI_RC_ERR_QUERY_LANGUAGE_NOT_SUPPORTED	Query Language not supported.
CMPI_RC_ERR_INVALID_QUERY	Invalid Query.
CMPI_RC_ERR_INVALID_HANDLE	Invalid encapsulated type handle.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

8.8 Threading Services

CMPI does not replace the native threading services of the underlying operating system. However, in order to enable garbage collection to execute reliably, CMPI should be notified when threads that are using CMPI services are created and destroyed. CMPI does provide a subset of threading primitives to allow the MI to be written in a platform-neutral fashion. While there is no limitation for a provider to use these functions, it is highly suggested that the MI use these threading primitives unless they need additional platform-specific functionality.

Threading is supported by the following functions.

CMPIBrokerFT.attachThread()

NAME

`CMPIBrokerFT.attachThread()` – inform the CMPI run-time system that the current thread will begin using CMPI services

SYNOPSIS

```
CMPIStatus CMPIBrokerFT.attachThread(
    const CMPIBroker* mb,
    const CMPIContext* ctx
);
```

DESCRIPTION

The `CMPIBrokerFT.attachThread()` function shall inform the CMPI run-time system that the current thread with the specified context will begin using CMPI services.

The `mb` argument points to a `CMPIBroker` structure.

The `ctx` argument points to the context object.

RETURN VALUE

The `CMPIBrokerFT.attachThread()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` code shall be recognized:

CMPI_RC_ERR_INVALID_HANDLE Invalid encapsulated type handle

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

`CMPIBrokerFT.detachThread()`, `CMPIBrokerFT.prepareAttachThread()`

CHANGE HISTORY

None.

CMPIBrokerFT.detachThread()

NAME

`CMPIBrokerFT.detachThread()` – inform the CMPI run-time system that the current thread will no longer be using CMPI services

SYNOPSIS

```
CMPIStatus CMPIBrokerFT.detachThread(
    const CMPIBroker* mb,
    const CMPIContext* ctx
);
```

DESCRIPTION

The `CMPIBrokerFT.dettachThread()` function shall inform the CMPI run-time system that the current thread will no longer be using CMPI services.

The `mb` argument points to a `CMPIBroker` structure.

The `ctx` argument points to the context object.

RETURN VALUE

The `CMPIBrokerFT.dettachThread()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` code shall be recognized:

CMPI_RC_ERR_INVALID_HANDLE Invalid encapsulated type handle.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerFT.prepareAttachThread()

NAME

`CMPIBrokerFT.prepareAttachThread()` – prepare the CMPI run-time system to accept a thread that will be using CMPI services

SYNOPSIS

```
CMPIContext* CMPIBrokerFT.prepareAttachThread(
    const CMPIBroker* mb,
    const CMPIContext* ctx
);
```

DESCRIPTION

The `CMPIBrokerFT.prepareAttachThread()` function shall prepare the CMPI run-time system to accept a thread that will be using CMPI services.

The `mb` argument points to a `CMPIBroker` structure.

The `ctx` argument points to the context object.

The returned `CMPIContext` object must be used by the subsequent `attachThread()` and `detachThread()` invocations.

RETURN VALUE

The `CMPIBrokerFT.prepareAttachThread()` function shall return a pointer to a `CMPIContext` structure to be used by the thread to be attached.

The following `CMPIrc` code shall be recognized:

CMPI_RC_ERR_INVALID_HANDLE Invalid encapsulated type handle.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

`CMPIBrokerFT.attachthread()`, `CMPIBrokerFT.detachThread()`

CHANGE HISTORY

None.

8.9 Operating System Abstractions

This set of OS functions abstraction services *must* be implemented by the MB. An MI can check the `CMPI_MB_OSEncapsulation` flag in `CMPI_BrokerFT.BrokerCaqpabilities` to ensure support.

Currently, this function table gives support for library name resolution and a subset of threading services. The CMPI threading services are based on the POSIX threading concepts. MB implementations shall ensure that these services will follow the POSIX threading semantics.

This does not imply that the underlying operating system must offer POSIX-conforming threading support. It only defines that threading services provided via this interface must map to POSIX threading semantics when the underlying operating system does not offer POSIX-conforming threading support natively.

Notice that these functions have no impact or relationship to any of the other CMP functions. Their sole purpose is to enable writing portable provider code (within limits) to be deployed across operating systems.

The `CMPI_THREAD_CDECL` is operating system-specific and should be defined appropriately for each operating system. For example, currently the macro on Windows is `__stdcall`, while under zOS it is defined as `__cdecl`.

```
#define CMPI_THREAD_TYPE void*
#define CMPI_THREAD_RETURN void*
#define CMPI_THREAD_KEY_TYPE void*
#define CMPI_MUTEX_TYPE void*
#define CMPI_COND_TYPE void*

typedef struct _CMPIBrokerExtFT {
    int ftVersion;
    char*(*resolveFileName)
        (const char* filename);

    CMPI_THREAD_TYPE (*newThread)
        (CMPI_THREAD_RETURN (CMPI_THREAD_CDECL* start)(void*
),
         void* parm, int detached);
    int (*joinThread)
        (CMPI_THREAD_TYPE thread,
         CMPI_THREAD_RETURN* retval );

    int (*exitThread)
        (CMPI_THREAD_RETURN return_code);
    int (*cancelThread)
        (CMPI_THREAD_TYPE thread);
    int (*threadSleep)
        (CMPIUint32 msec);
    int (*threadOnce)
        (int* once, void (*init)(void));

    int (*createThreadKey)
        (CMPI_THREAD_KEY_TYPE* key, void (*cleanup)(void*));
```

```
    int (*destroyThreadKey)
        (CMPI_THREAD_KEY_TYPE key);
    void* (*getThreadSpecific)
        (CMPI_THREAD_KEY_TYPE key);
    int (*setThreadSpecific)
        (CMPI_THREAD_KEY_TYPE key, void*  value);

    CMPI_MUTEX_TYPE (*newMutex)
        (int opt);
    void (*destroyMutex)
        (CMPI_MUTEX_TYPE);
    void (*lockMutex)
        (CMPI_MUTEX_TYPE);
    void (*unlockMutex)
        (CMPI_MUTEX_TYPE);

    CMPI_COND_TYPE (*newCondition)
        (int opt);
    void (*destroyCondition)
        (CMPI_COND_TYPE);
    int (*condWait)
        (CMPI_COND_TYPE cond, CMPI_MUTEX_TYPE mutex);
    int (*timedCondWait)
        (CMPI_COND_TYPE cond, CMPI_MUTEX_TYPE mutex,
         struct timespec* wait);
    int (*signalCondition)
        (CMPI_COND_TYPE cond);
} CMPIBrokerExtFT;
```

CMPIBrokerExtFT.resolveFileName()

NAME

`CMPIBrokerExtFT.resolveFileName()` – complement a generic dynamic library name to its OS-dependent native format

SYNOPSIS

```
char* CMPIBrokerExtFT.resolveFileName(
    const char* libName
);
```

DESCRIPTION

The `CMPIBrokerExtFT.resolveFileName()` function shall complement a generic dynamic library name to its OS-dependent native format.

The `libName` argument points to the generic library name.

RETURN VALUE

The returned `char*` pointer points to the library name in native OS format. This memory does not need to be freed as it is tracked internally by CMPI.

ERRORS

In case no storage could be obtained for the complemented library name, NULL will be returned.

EXAMPLES

None.

APPLICATION USAGE

Depending on the underlying OS, dynamic library names have specific formats. The native format of "`MyLibrary`" on Linux for this name is "`libMyLibrary.so`". The native format on Windows is "`MyLibrary.dll`". This service performs the mapping of library names.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerExtFT.newThread()

NAME

`CMPIBrokerExtFT.newThread()` – start a new thread using POSIX threading semantics

SYNOPSIS

```
CMPI_THREAD_TYPE CMPIBrokerExtFT.newThread(
    CMPI_THREAD_RETURN (CMPI_THREAD_CDECL* start) (void*),
    void* parm,
    int detached
);
```

DESCRIPTION

The `CMPIBrokerExtFT.newThread()` function shall start a new thread using POSIX threading semantics.

The `start` argument is a pointer to the function to be started as a thread.

The `parm` argument is a pointer to parameter(s) to be passed to the thread.

The `detached` argument, if not zero, defines that the thread should run in detached mode.

RETURN VALUE

The return value is the thread ID.

ERRORS

In case of error, NULL will be returned.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerExtFT.joinThread()

NAME

`CMPIBrokerExtFT.joinThread()` – wait until the specified thread ends using POSIX threading semantics

SYNOPSIS

```
int CMPIBrokerExtFT.joinThread(
    CMPI_THREAD_TYPE thread,
    CMPI_THREAD_RETURN* retval,
);
```

DESCRIPTION

The `CMPIBrokerExtFT.joinThread()` function shall wait until the specified thread ends using POSIX threading semantics.

The `thread` argument is the thread ID of the thread waiting for completion.

The `retval` argument is a pointer to the return value of the thread.

RETURN VALUE

Completion code as defined by POSIX threading specifications.

ERRORS

As defined by POSIX threading specifications.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerExtFT.cancelThread()

NAME

`CMPIBrokerExtFT.cancelThread()` – cancel a running thread using POSIX threading semantics

SYNOPSIS

```
int CMPIBrokerExtFT.cancelThread(
    CMPI_THREAD_TYPE thread
);
```

DESCRIPTION

The `CMPIBrokerExtFT.cancelThread()` function shall cancel a thread identified by the `thread` parameter using POSIX threading semantics.

The `thread` argument represents the thread to be canceled.

RETURN VALUE

Completion code as defined by POSIX threading specifications.

ERRORS

As defined by POSIX threading specifications.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerExtFT.exitThread()

NAME

`CMPIBrokerExtFT.exitThread()` – cause the current thread to exist with the passed-in return code using POSIX threading semantics

SYNOPSIS

```
int CMPIBrokerExtFT.exitThread(
    CMPI_THREAD_RETURN return_code
)
```

DESCRIPTION

The `CMPIBrokerExtFT.exitThread()` function causes the current thread to exit with the passed-in return code using POSIX threading semantics.

The `return_code` argument is the return code that should be used for the thread.

RETURN VALUE

The `CMPIBrokerExtFT.exitThread()` function never returns.

ERRORS

As defined by POSIX threading specifications.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

Added in CMPI 2.0.

CMPIBrokerExtFT.threadSleep()

NAME

`CMPIBrokerExtFT.threadSleep()` – suspend execution of the current thread for a specified duration

SYNOPSIS

```
int CMPIBrokerExtFT.threadSleep(
    CMPIUint32 msec
);
```

DESCRIPTION

The `CMPIBrokerExtFT.threadSleep()` function shall suspend execution of the current thread for a specified duration.

The `msec` argument specifies the suspend duration in milliseconds.

RETURN VALUE

Completion code as defined by POSIX threading specifications.

ERRORS

As defined by POSIX threading specifications.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerExtFT.threadOnce()

NAME

`CMPIBrokerExtFT.threadOnce()` – ensure execution of the specified function procedure only once during the lifetime of the thread

SYNOPSIS

```
int CMPIBrokerExtFT.threadOnce(
    (int* once, void (*function)(void)
);
```

DESCRIPTION

The `CMPIBrokerExtFT.threadOnce()` function shall ensure execution of the specified function procedure only once during the lifetime of the thread.

The `function` argument specifies the function to be invoked.

RETURN VALUE

Completion code as defined by POSIX threading specifications.

ERRORS

As defined by POSIX threading specifications.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerExtFT.createThreadKey()

NAME

`CMPIBrokerExtFT.createThreadKey()` – create a POSIX threading-conformant thread key

SYNOPSIS

```
int CMPIBrokerExtFT.createThreadKey(
    CMPI_THREAD_KEY_TYPE* key,
    void (*cleanup)(void*)
);
```

DESCRIPTION

The `CMPIBrokerExtFT.createThreadKey()` function shall create a POSIX threading-conformant thread key that can be used as a key to access the thread local store.

The `key` argument specifies the address for the key to be returned.

The `cleanup` argument specifies the function to be invoked during thread local store cleanup.

RETURN VALUE

Completion code as defined by POSIX threading specifications.

ERRORS

As defined by POSIX threading specifications.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerExtFT.destroyThreadKey()

NAME

`CMPIBrokerExtFT.destroyThreadKey()` – destroy a POSIX threading-conformant thread key

SYNOPSIS

```
int CMPIBrokerExtFT.destroyThreadKey(
    CMPI_THREAD_KEY_TYPE key
);
```

DESCRIPTION

The `CMPIBrokerExtFT.destroyThreadKey()` function shall destroy a POSIX threading-conformant thread key.

The `key` argument specifies the key to be destroyed.

RETURN VALUE

Completion code as defined by POSIX threading specifications.

ERRORS

As defined by POSIX threading specifications.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerExtFT.getThreadSpecific()

NAME

`CMPIBrokerExtFT.getThreadSpecific()` – get data from the thread local store using a thread key

SYNOPSIS

```
void* CMPIBrokerExtFT.getThreadSpecific(
    CMPI_THREAD_KEY_TYPE key
);
```

DESCRIPTION

The `CMPIBrokerExtFT.getThreadSpecific()` function shall return data from the thread local store using a thread key.

The `key` argument specifies the key to be used to retrieve the data.

RETURN VALUE

Pointer to data found or NULL in case data is not found using this key.

ERRORS

NULL is returned if data is not found using this key.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerExtFT.setThreadSpecific()

NAME

`CMPIBrokerExtFT.setThreadSpecific()` – set a pointer to data in the thread local store using a thread key

SYNOPSIS

```
void* CMPIBrokerExtFT.setThreadSpecific(
    CMPI_THREAD_KEY_TYPE key,
    void* data
);
```

DESCRIPTION

The `CMPIBrokerExtFT.setThreadSpecific()` function shall accept a pointer to data and place it in the thread local store using a thread key.

The `key` argument specifies the key to be used.

The `data` argument specifies the pointer to the data.

RETURN VALUE

Completion code as defined by POSIX threading specifications.

ERRORS

As defined by POSIX threading specifications.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerExtFT.newMutex()

NAME

`CMPIBrokerExtFT.newMutex()` – create a POSIX threading-conformant mutex

SYNOPSIS

```
CMPI_MUTEX_TYPE CMPIBrokerExtFT.newMutex(
    int opt
);
```

DESCRIPTION

The `CMPIBrokerExtFT.newMutex()` function shall return a handle to a new POSIX-conforming mutex.

The `opt` argument specifies POSIX options to be used. If no options are to be defined, the 0 values must be used.

RETURN VALUE

Handle of newly created mutex.

ERRORS

As defined by POSIX threading specifications.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerExtFT.destroyMutex()

NAME

`CMPIBrokerExtFT.destroyMutex()` – destroy a POSIX threading-conformant mutex

SYNOPSIS

```
void CMPIBrokerExtFT.destroyMutex(
    CMPI_MUTEX_TYPE mutex
);
```

DESCRIPTION

The `CMPIBrokerExtFT.destroyMutex()` function shall destroy a POSIX-conforming mutex.

The `mutex` argument specifies the mutex to be destroyed.

RETURN VALUE

None.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerExtFT.lockMutex()

NAME

`CMPIBrokerExtFT.lockMutex()` – request control of a mutex

SYNOPSIS

```
void CMPIBrokerExtFT.lockMutex(
    CMPI_MUTEX_TYPE mutex
);
```

DESCRIPTION

The `CMPIBrokerExtFT.lockMutex()` function shall attempt to get control of the mutex and must wait until released when not available.

The `mutex` argument specifies the mutex to be locked.

RETURN VALUE

None.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerExtFT.unlockMutex()

NAME

`CMPIBrokerExtFT.unlockMutex()` – release control of a mutex

SYNOPSIS

```
void CMPIBrokerExtFT.unlockMutex(
    CMPI_MUTEX_TYPE mutex
);
```

DESCRIPTION

The `CMPIBrokerExtFT.unlockMutex()` function shall release control of the mutex.

The `mutex` argument specifies the mutex to be unlocked.

RETURN VALUE

None.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerExtFT.newCondition()

NAME

`CMPIBrokerExtFT.newCondition()` – create a new POSIX threading-conformant condition variable

SYNOPSIS

```
CMPI_COND_TYPE CMPIBrokerExtFT.newCondition(
    int opt
);
```

DESCRIPTION

The `CMPIBrokerExtFT.newCondition()` function shall return a handle to a new POSIX-conforming condition variable.

The `opt` argument specifies POSIX options to be used. If no options are to be defined, the 0 values must be used.

RETURN VALUE

Handle of newly created condition variable.

ERRORS

As defined by POSIX threading specifications.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerExtFT.destroyCondition()

NAME

`CMPIBrokerExtFT.destroyCondition()` – destroy a condition variable

SYNOPSIS

```
void CMPIBrokerExtFT.destroyCondition(
    CMPI_COND_TYPE condition
);
```

DESCRIPTION

The `CMPIBrokerExtFT.destroyCondition()` function shall destroy a condition variable.

The `condition` argument specifies the handle of the condition variable to be destroyed.

RETURN VALUE

None.

ERRORS

As defined by POSIX threading specifications.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerExtFT.condWait()

NAME

`CMPIBrokerExtFT.condWait()` – wait until the condition is signalled

SYNOPSIS

```
void CMPIBrokerExtFT.condWait(
    CMPI_COND_TYPE cond,
    CMPI_MUTEX_TYPE mutex
);
```

DESCRIPTION

The `CMPIBrokerExtFT.condWait()` function shall return when `condition` has been signalled already and otherwise must wait for the signal and then return.

The `condition` argument specifies the handle of the condition variable to be used.

The `mutex` argument specifies the handle of a locked mutex guarding this condition variable.

RETURN VALUE

As defined by POSIX threading specifications.

ERRORS

As defined by POSIX threading specifications.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerExtFT.timedCondWait()

NAME

`CMPIBrokerExtFT.timedCondWait()` – wait until the condition is signalled using a timeout value

SYNOPSIS

```
void CMPIBrokerExtFT.timedCondWait(
    CMPI_COND_TYPE cond,
    CMPI_MUTEX_TYPE mutex,
    struct timespec* wait
);
```

DESCRIPTION

The `CMPIBrokerExtFT.timedCondWait()` function shall return when `condition` has been signalled already and otherwise must wait for the signal and then return. The function shall return when the timeout specification elapses before the condition is signalled.

The `condition` argument specifies the handle of the condition variable to be used.

The `mutex` argument specifies the handle of a locked mutex guarding this condition variable.

The `wait` argument specifies the timeout value.

RETURN VALUE

As defined by POSIX threading specifications.

ERRORS

As defined by POSIX threading specifications.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerExtFT.signalCondition()

NAME

`CMPIBrokerExtFT.signalCondition()` – send a signal to a condition variable

SYNOPSIS

```
void CMPIBrokerExtFT.signalCondition(
    CMPI_COND_TYPE cond
);
```

DESCRIPTION

The `CMPIBrokerExtFT.signalCondition()` function shall send a signal to a condition variable.

The `condition` argument specifies the handle of the target condition variable.

RETURN VALUE

As defined by POSIX threading specifications.

ERRORS

As defined by POSIX threading specifications.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

8.10 Memory Enhancements

This set of memory enhancements functions are optional and available only when the `CMPI_MB_MemoryEnhancmentSupport` flag in `CMPIBrokerFT.brokerCapabilities` is set. `CMPIBrokerMemFT` is routed in `CMPIBrokerFT`.

The functions in `CMPIBrokerMemFT` are used to manipulate memory. They are used for garbage collection, allocation, and deallocation of memory. These do not replace the "clone" and "release" functions defined in the various CMPI data structures; those should continue to be used as previously documented. These new functions allow providers to participate more openly in the garbage collection mechanism provided by the MB, and to keep non-CMPI-related memory allocated via this mechanism. An MB does not need to actually provide anything except a pass through to the underlying OS call. All MIs should use these functions for memory management.

There are three sets of memory related functions and each can help in various situations. The "mark" and "release" functions can be used by long-running secondary threads. The `cmpiMalloc`, `cmpiCalloc`, `cmpiRealloc`, and `cmpiStrDup` functions can be used to hook-up the providers memory data storage to the MB garbage collector. The "freeXXX" functions can be used to instruct the garbage collector that the CMPI objects are no longer needed and can be collected.

Long-running secondary threads are a common situation in indication providers. They often stay for the life-time of the MB and often create new `CMPIInstance` objects. Unfortunately, those CMPI objects are not being garbage collected because the thread has not finished executing. The "mark" and "release" functions help in this fashion by setting a marker which the garbage collector can use to determine which of CMPI objects are temporary. When the "release" function is called, the garbage collector kicks in and reclaims all of the CMPI objects that have been created since the "mark" call.

The garbage collector can also be used to support the user-defined data storage. By using the `cmpiMalloc`, `cmpiCalloc`, `cmpiRealloc`, and `cmpiStrDup` functions, the resulting data storage is included in the garbage collector and when it is invoked, those data storages become automatically reclaimed.

```
typedef struct CMPIBrokerMemFT {
    const int ftVersion;

        CMPIGcStat* (*mark)(const CMPIBroker*, CMPIStatus*);
        CMPIStatus (*release)(const CMPIBroker*,
            const CMPIGcStat*);
        void* (*cmpiMalloc)(const CMPIBroker*, size_t);
        void* (*cmpiCalloc)(const CMPIBroker*, size_t,
            size_t);
        void* (*cmpiRealloc)(const CMPIBroker*, void*,
            size_t);
        char* (*cmpiStrDup)(const CMPIBroker*, const char*);
        void (*cmpiFree)(const CMPIBroker*, void*);
        void (*freeInstance)(const CMPIBroker*,
            CMPIInstance*);
        void (*freeObjectPath)(const CMPIBroker*,
            CMPIObjectPath*);
```

```
        void (*freeArgs)(const CMPIBroker*, CMPIArgs*);
        void (*freeString)(const CMPIBroker*, CMPIString*);
        void (*freeArray)(const CMPIBroker*, CMPIArray*);
        void (*freeDateTime)(const CMPIBroker*,
            CMPIDateTime*);
        void (*freeSelectExp)(const CMPIBroker*,
            CMPISelectExp*);
} CMPIBrokerMemFT;
```

CMPIBrokerMemFT.cmpiCalloc()

NAME

`CMPIBrokerMemFT.cmpiCalloc()` – allocate memory of the specified size; memory is initialized to zero

SYNOPSIS

```
void* CMPIBrokerMemFT.cmpiCalloc(
    const CMPIBroker* mb,
    size_t nElems,
    size_t sizeElem
);
```

DESCRIPTION

This function shall return a pointer to the allocated memory; the memory will be initialized to zero.

The `mb` argument specifies the broker.

The `nElems` argument specifies the number of elements to allocate.

The `sizeElem` specifies the number of elements to allocate.

RETURN VALUE

Returns a pointer to the allocated memory location, or NULL if memory could not be allocated.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerMemFT.cmpiFree()

NAME

`CMPIBrokerMemFT.cmpiFree()` – indicate to the MB that the memory associated with the passed-in pointer can be reclaimed by the garbage collector

SYNOPSIS

```
void CMPIBrokerMemFT.cmpiFree(
    const CMPIBroker* mb,
    void* ptr
);
```

DESCRIPTION

This function frees memory allocated via the `cmpiMalloc`, `cmpiCalloc`, or `cmpiRealloc` functions.

The `mb` argument specifies the broker.

The `ptr` argument specifies the memory to free. This memory *must* have been allocated via the `cmpiMalloc`, `cmpiCalloc`, or `cmpiRealloc` functions.

RETURN VALUE

None.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerMemFT.cmpiFreeInstance()

NAME

`CMPIBrokerMemFT.cmpiFreeInstance()` – indicate to the MB that the memory associated with the passed-in pointer can be reclaimed by the garbage collector

SYNOPSIS

```
void CMPIBrokerMemFT.freeInstance(
    const CMPIBroker* mb,
    CMPIInstance* inst
);
```

DESCRIPTION

Allows an MI to free memory associated to a `CMPIinstance` which was allocated via `CMPIBrokerEncFT.newInstance`. This function should be called when an instance is no longer being used by the MI. This function will free all contained objects (e.g., properties).

RETURN VALUE

None.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerMemFT.cmpiMalloc()

NAME

`CMPIBrokerMemFT.cmpiMalloc()` – allocate uninitalized memory of the specified size

SYNOPSIS

```
void* CMPIBrokerMemFT.cmpiMalloc(
    const CMPIBroker* mb,
    size_t size
);
```

DESCRIPTION

This function shall return a pointer to the allocated memory.

The `mb` argument specifies the broker.

The `size` argument specifies the amount of memory to allocate.

RETURN VALUE

Returns a pointer to the allocated memory location, or NULL if memory could not be allocated.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerMemFT.cmpiRealloc()

NAME

`CMPIBrokerMemFT.cmpiRealloc()` – change the size of the memory block to the passed-in size

SYNOPSIS

```
CMPIBrokerMemFT.cmpiRealloc(
    const CMPIBroker* mb,
    void* ptr,
    size_t size
);
```

DESCRIPTION

This function changes the size of the memory block pointed to by `ptr` which must have been returned by a previous call to `cmpiMalloc` or `cmpiCalloc`. See the ANSI-C function *realloc*() for more information.

The `mb` argument specifies the broker.

The argument `ptr` is a pointer to previosuly allocated memory. Passing a pointer to this function which was not allocated explicitly by `cmpiMalloc` or `cmpiCalloc` is undefined.

The `size` argument specifies the new size of the memory block.

RETURN VALUE

Returns a pointer to the newly allocated memory block, or NULL if the new memory is not allocated. If the function fails nothing is done with the original `ptr` argument.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerMemFT.freeArgs()

NAME

`CMPIBrokerMemFT.freeArgs()` – indicate to the MB that the passed-in `CMPIArgs` can be released

SYNOPSIS

```
void CMPIBrokerMemFT.freeArgs(
    const CMPIBroker* mb,
    CMPIArgs* args
)
```

DESCRIPTION

Allows an MI to free memory associated to a `CMPIArgs` which was allocated via `CMPIBrokerEncFT.newArgs`. This function should be called when an instance is no longer being used by the MI. This function will free all contained objects.

RETURN VALUE

None

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerMemFT.freeArray()

NAME

`CMPIBrokerMemFT.freeArray()` – indicate to the MB that the passed-in `CMPIArray` can be released

SYNOPSIS

```
void CMPIBrokerMemFT.freeArray(
    const CMPIBroker* mb,
    CMPIArray* string
);
```

DESCRIPTION

Allows an MI to free memory associated to a `CMPIArray` which was allocated via `CMPIBrokerEncFT.newArray`. This function should be called when an instance is no longer being used by the MI. This function will free all contained objects (e.g., the array elements).

RETURN VALUE

None.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerMemFT.freeDateTime()

NAME

`CMPIBrokerMemFT.freeDateTime()` – indicate to the MB that the passed-in `CMPIDateTime` can be released

SYNOPSIS

```
void CMPIBrokerMemFT.freeDateTime(
    const CMPIBroker* mb,
    CMPIDateTime* dt
);
```

DESCRIPTION

Allows an MI to free memory associated to a `CMPIDateTime` which was allocated via the `CMPIBrokerEncFT.newDateTime` functions. This function should be called when an instance is no longer being used by the MI. This function will free all contained objects.

RETURN VALUE

None.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerMemFT.freeObjectPath()

NAME

`CMPIBrokerMemFT.freeObjectPath()` – indicate to the MB that the passed-in `CMPIObjectPath` can be released

SYNOPSIS

```
void CMPIBrokerMemFT.freeObjectPath(
    const CMPIBroker* mb,
    CMPIObjectPath* obj
);
```

DESCRIPTION

Allows an MI to free memory associated to a `CMPIObjectPath` which was allocated via `CMPIBrokerEncFT.newObjectPath`. This function should be called when an instance is no longer being used by the MI. This function will free all contained objects (e.g., properties).

RETURN VALUE

None.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerMemFT.freeSelectExp()

NAME

`CMPIBrokerMemFT.freeSelectExp()` – indicate to the MB that the passed-in `CMPISelectExp` can be released

SYNOPSIS

```
void CMPIBrokerMemFT.freeSelectExp(
    const CMPIBroker* mb,
    CMPISelectExp* dt
);
```

DESCRIPTION

Allows an MI to free memory associated to a `CMPISelectExp` which was allocated via the `CMPIBrokerEncFT.newSelectExp` function. This function should be called when an instance is no longer being used by the MI. This function will free all contained objects.

RETURN VALUE

None.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerMemFT.freeString()

NAME

`CMPIBrokerMemFT.freeString()` – indicate to the MB that the passed-in `CMPIString` can be released

SYNOPSIS

```
void CMPIBrokerMemFT.freeString(
    const CMPIBroker* mb,
    CMPIString* string
);
```

DESCRIPTION

Allows an MI to free memory associated to a `CMPIString` which was allocated via `CMPIBrokerEncFT.newString`. This function should be called when an instance is no longer being used by the MI. This function will free all contained objects.

RETURN VALUE

None.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerMemFT.mark()

NAME

`CMPIBrokerMemFT.mark()` – set a marker for subsequent newly created CMPI objects

SYNOPSIS

```
CMPIGcStat* (*mark)(
    const CMPIBroker* mb,
    CMPIStatus* rc
);
```

DESCRIPTION

Invoking this function marks subsequent newly created CMPI objects to be released when the `release()` function is invoked. The return value is used by the `release()` function to find which CMPI structures since the `mark()` function can be de-allocated.

Note that `mark()` functions can be stacked.

The `mb` argument specifies the broker.

The `rc` argument provides the return status (suppressed when NULL).

RETURN VALUE

This function returns a handle which is to be provided to the `release()` function.

ERRORS

The `CMPIBrokerMemFT.mark()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

CMPIBrokerMemFT.release()

NAME

`CMPIBrokerMemFT.release()` – release all CMPI objects created since last `mark()` operation represented by the parameter

SYNOPSIS

```
CMPIStatus (*release)(const CMPIBroker* mb,
    const CMPIGcStat* gc
);
```

DESCRIPTION

This function releases all CMPI objects created since the last `mark()` operation represented by the parameter.

Note that `release()` function calls can be stacked.

The `mb` argument specifies the broker.

The `gc` argument is the handle returned from the `mark()` operation.

RETURN VALUE

The `CMPIBrokerMemFT.release()` function shall return a `CMPIStatus` structure containing the service return status.

The following `CMPIrc` codes shall be recognized:

CMPI_RC_OK	Operation successful.
CMPI_RC_ERR_FAILED	Unspecific error occurred.

ERRORS

None.

EXAMPLES

None.

APPLICATION USAGE

None.

SEE ALSO

None.

CHANGE HISTORY

None.

A Examining a Query using CMPI

MIs may choose to examine the WHERE clause to see if any of the instances that the MI manages satisfies the WHERE clause. Use-cases include an MI's implementation of indications processing, or `execQuery()`:

```
CMPIStatus CMPIInstanceMIFT.execQuery(CMPIInstanceMI* mi
    const CMPIContext* ctx,
    const CMPIResult* rslt,
    const CMPIObjectPath* op,
    const char* query,
    const char* lang);
```

For an MI to use a raw query string it is imaginably a daunting task for the MI developer, requiring elements such as a text parser and boolean logic processor. This specification defines broker functions that MI developers may use to simplify the complexities of discerning the applicability of their instances. An MI developer may want to know what properties are asked for, or what elements comprise the WHERE clause, in order to efficiently marshal MI resources. A provider may need to contact or start certain services depending on the CIM classes or properties referred to.

In order to leverage the broker query processing functionality, `CMPIBroker.brokerCapabilities` must have the `CMPI_MB_QueryNormalization` flag present. If this flag is enabled, the following structures are available to the MI: `CMPISelectExp`, `CMPISelectCond`, `CMPISubCond`, and `CMPIPredicates`. The implementation may place restrictions on what functions are supported in each structure's function table, by returning CMPI_RC_ERR_NOT_SUPPORTED.

A.1 CMPISelectExp

Given a query language and query expression, a `CMPISelectExp` may be created to expose the parse tree from the WHERE clause, and the requested properties from the SELECT clause:

```
CMPISelectExp* CMPIBrokerEncFT.newSelectExp(
    const CMPIBroker* mb,
    const char* query,
    const char* lang,
    const CMPIArray** projection,
    CMPIStatus* rc);
```

The `lang` parameter specifies the query language in which `query` is to be interpreted. The following languages strings are recognized by the DMTF:

`"DMTF:CQL"` The CQL standard as defined by DSP0202.

`"WQL"` A commonly practiced predecessor to CQL.

The `projection` output parameter is populated with `CMPIStrings` containing the properties of the `query` parameter's SELECT clause. The array elements may be accessed using the `CMPIArrayFT` functions. Note that the query language may enforce language-specific syntax (such as chains) on the elements of the `projection` array. `CMPIString` elements of the `projection` parameter may have the following syntax:

```
DMTF:CQL  CIM_LogicalDevice::OperationalStatus
WQL:      OperationalStatus
```

Providers may need to check for language-specific syntax in these elements when reading the results of the `projection` array.

The returned `CMPISelectExp` has the following functions to access elements in the WHERE clause parse tree:

```
CMPISelectExpFT
CMPISelectCond* getDOC(const CMPISelectExp*
    exp,CMPIStatus* status);
CMPISelectCond* getCOD(const CMPISelectExp*
    exp,CMPIStatus* status);
```

A.2 CMPISelectCond and Normalized Form

Given a `CMPISelectExp` structure, the `getDOC()` and `getCOD()` functions return one `CMPISelectCond` representing a WHERE clause normalized into one of two forms: a “Conjunction of Disjunctions” or “Disjunction of Conjunctions”.

A Conjunction of Disjunctions (COD) or Conjunctive Normal Form (CNF) is a group of OR'ed terms in parentheses which are AND'ed together:

```
(X OR Y) AND (X OR NOT A OR B) AND (Z) AND (NOT W OR X)
```

A Disjunction of Conjunctions (DOC) or Disjunctive Normal Form (DNF) is a group of AND'ed terms in parentheses which are OR'ed together:

```
(NOT X AND Y AND Z) OR (NOT A) OR (B AND C AND D)
```

In both cases, the only operators allowed in a COD/DOC representations are AND, OR, and NOT. The NOT operator can only be applied to a term, not an entire conjunction or disjunction. The DOC and COD forms do not require that every conjunction (AND'ed terms) or disjunction (OR'ed terms) be strictly binary. Every WHERE clause can be represented in DOC or COD form. However, the DOC and COD form is not guaranteed to have fewer terms then the original WHERE clause. The benefit to COD and DOC form is a layout of boolean expressions which are grouped in a predictable format.

Consider the following examples of COD and DOC representations of a WHERE clause:

Example 1: WHERE A=1 AND B=2

```
COD: (A=1) AND (B=2)
DOC: (a=1 AND B=2)
```

Example 2: WHERE A=1 AND B=2 OR C=3

```
COD: (A=1 OR C=3) AND (B=2 OR C=3)
DOC: (A=1 AND B=2) OR (c=3)
```

A.3 CMPISubCond

Once either a COD or DOC is chosen, the resulting `CMPISelectCond` function table includes a means to discern the form represented, and how many conjunctions or disjunctions exist. Each conjunction or disjunction can be accessed directly, and dissected further:

```
CMPISubCondFT
CMPICount getCountAndType(const CMPISelectCond* cond,
    int* type,
    CMPIStatus* status);

CMPISubCond* getSubCondAt(const CMPISelectCond* cond,
    CMPICount which,
    CMPIStatus* status);
```

Suppose `CMPISelectCond` represents a COD form for the WHERE clause in Example 2:

```
(A=1 OR C=3) AND (B=2 OR C=3)
```

An invocation of `getCountAndType()` will set the type output parameter to 1 (0 means DOC), and return a count of 2 (there are two terms AND'ed together making up the conjunction). The `getSubCondAt()` `function` can be used to fetch one of two possible `CMPISubCond` structures, representing a disjunction clause in parentheses:

```
GetSubCondAt(0)   GetSubCondAt(1)
(A=1 OR C=3)     (B=2 OR C=3)
```

A.4 CMPIPredicate

The `CMPISubCond()` function table allows identification of the tokens and values comprising the sub-conditions. This is encapsulated as a `CMPIPredicate`:

```
CMPISubCondFT
CMPICount getCount(const CMPISubCond* cond,
    CMPIStatus* status);

CMPIPredicate* getPredicateAt(const CMPISubCond* cond,
    CMPICount which,
    CMPIStatus* status);

CMPIPredicate* getPredicate(const CMPISubCond* cond,
    const char* name,
    CMPIStatus* status);
```

`CMPIPredicates` refer to the boolean expressions inside each conjunction or disjunction represented by `CMPISubCond`. Continuing with the COD expression of Example 2, the terms `A=1`, `C=3`, `B=2`, and `C=3` are predicates. Predicates may be retrieved by their order in the `CMPISubCond`, or by name using the `getPredicate()` function. The name of the predicate is the symbol on the left-hand side: The predicate `A=1` is named “A”. It is possible to have more than one predicate by the same name.

A.5 CMPIPredOp and the Logical Operands

Predicates can be dissected further into the left-hand value, right-hand value, and the binding logical operation:

```
CMPIStatus getData(const CMPIPredicate* pred,
    CMPIType* type,
    CMPIPredOp* op,
    CMPIString** left,
    CMPIString** right);
```

The `getData()` function allows identification of the left and right-hand tokens and the operation that relates them (see the `CMPIPredOp` enumeration). The `type` output parameter refers to the `CIMType` property type or "units" of the logical operation.

A.6 Relationship for CMPI Normalization Structures

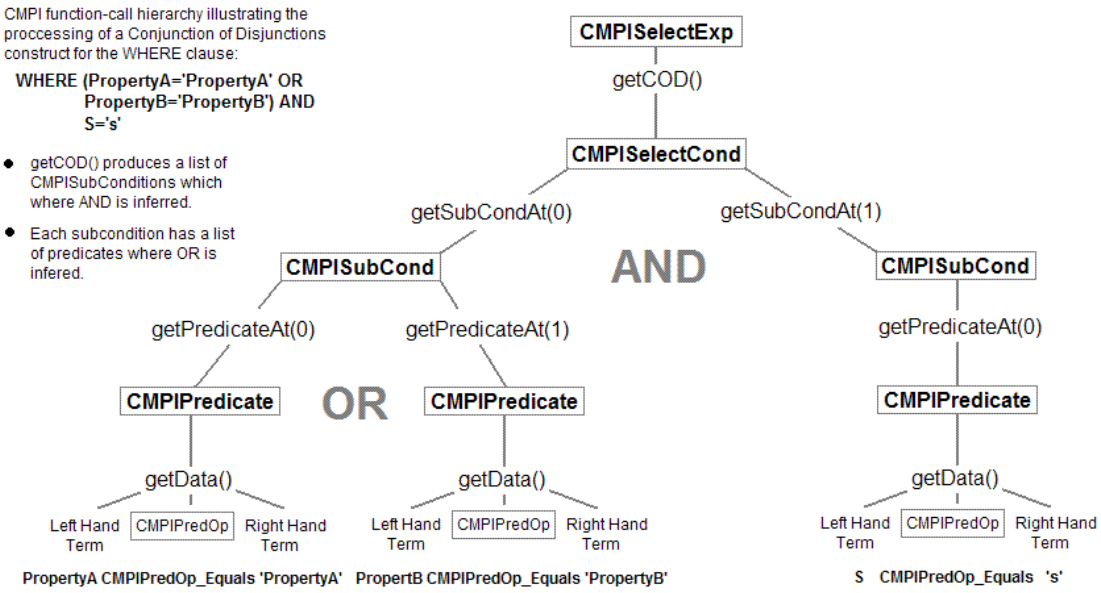


Figure 8-1: Relation of the CMPI Normalization Structures

A.7 Bibliography Addenda

- http://en.wikipedia.org/wiki/Conjunctive_normal_form
- http://en.wikipedia.org/wiki/Disjunctive_normal_form

Index