ISO/IEC 11002

Edition 1.0 2008-07

INTERNATIONAL STANDARD

Information technology – Multipath management (API)

IEC Central Office
3, rue de Varembé
CH-1211 Geneva 20
Switzerland
Email: inmail@iec.ch
Web: www.iec.ch

About the IEC

The International Electrotechnical Commission (IEC) is the leading global organization that prepares and publishes International Standards for all electrical, electronic and related technologies.

About IEC publications

The technical content of IEC publications is kept under constant review by the IEC. Please make sure that you have the latest edition, a corrigenda or an amendment might have been published.

- Catalogue of IEC publications: www.iec.ch/searchpub
The IEC on-line Catalogue enables you to search by a variety of criteria (reference number, text, technical committee,…). It also gives information on projects, withdrawn and replaced publications.

- IEC Just Published: www.iec.ch/online_news/justpub
Stay up to date on all new IEC publications. Just Published details twice a month all new publications released. Available on-line and also by email.

- Electropedia: www.electropedia.org
The world's leading online dictionary of electronic and electrical terms containing more than 20 000 terms and definitions in English and French, with equivalent terms in additional languages. Also known as the International Electrotechnical Vocabulary online.

- Customer Service Centre: www.iec.ch/webstore/custserv
If you wish to give us your feedback on this publication or need further assistance, please visit the Customer Service Centre FAQ or contact us:

Email: csc@iec.ch
Tel.: +41 22 919 02 11
Fax: +41 22 919 03 00

ISO/IEC 11002

Edition 1.0 2008-07

INTERNATIONAL STANDARD

Information technology – Multipath management (API)

INTERNATIONAL
ELECTROTECHNICAL
COMMISSION

PRICE CODE W

ICS 35.200

ISBN 2-8318-9908-7

CONTENTS

INFORMATION TECHNOLOGY – MULTIPATH MANAGEMENT (API)

FOREWORD

1) ISO (International Organization for Standardization) and IEC (International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards. Their preparation is entrusted to technical committees; any ISO and IEC member body interested in the subject dealt with may participate in this preparatory work. International governmental and non-governmental organizations liaising with ISO and IEC also participate in this preparation.

2) In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

3) The formal decisions or agreements of IEC and ISO on technical matters express, as nearly as possible, an international consensus of opinion on the relevant subjects since each technical committee has representation from all interested IEC and ISO member bodies.

4) IEC, ISO and ISO/IEC publications have the form of recommendations for international use and are accepted by IEC and ISO member bodies in that sense. While all reasonable efforts are made to ensure that the technical content of IEC, ISO and ISO/IEC publications is accurate, IEC or ISO cannot be held responsible for the way in which they are used or for any misinterpretation by any end user.

5) In order to promote international uniformity, IEC and ISO member bodies undertake to apply IEC, ISO and ISO/IEC publications transparently to the maximum extent possible in their national and regional publications. Any divergence between any ISO/IEC publication and the corresponding national or regional publication should be clearly indicated in the latter.

6) ISO and IEC provide no marking procedure to indicate their approval and cannot be rendered responsible for any equipment declared to be in conformity with an ISO/IEC publication.

7) All users should ensure that they have the latest edition of this publication.

8) No liability shall attach to IEC or ISO or its directors, employees, servants or agents including individual experts and members of their technical committees and IEC or ISO member bodies for any personal injury, property damage or other damage of any nature whatsoever, whether direct or indirect, or for costs (including legal fees) and expenses arising out of the publication of, use of, or reliance upon, this ISO/IEC publication or any other IEC, ISO or ISO/IEC publications.

9) Attention is drawn to the normative references cited in this publication. Use of the referenced publications is indispensable for the correct application of this publication.

10) Attention is drawn to the possibility that some of the elements of this International Standard may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

International Standard ISO/IEC 11002 was prepared by the Information Technology Industry Council and was adopted under the fast track procedure by subcommittee 25: Interconnection of information technology equipment, of ISO/IEC joint technical committee 1: Information technology.

This International Standard has been approved by vote of the member bodies, and the voting results may be obtained from the address given on the second title page.

This publication has been drafted in accordance with the ISO/IEC Directives, Part 2.

INTRODUCTION

The Multipath Management application programming interface (API) provides management interfaces to standard capabilities defined in ISO/IEC 14776-453 (SPC-3) and common vendor-specific extensions to the standard capabilities. The intended audience is vendors that deliver drivers that provide these capabilities. This standard relates to SCSI multipathing features and excludes multipathing between interconnect devices (such as Fibre Channel switches) and transport specific multipathing (such as iSCSI multiple connections per session).

INFORMATION TECHNOLOGY – MULTIPATH MANAGEMENT API

1 Scope

This International Standard provides management interfaces to standard capabilities defined in ISO/IEC 14776-453 (SPC-3) and common vendor-specific extensions to the standard capabilities. The intended audience is vendors that deliver drivers that provide these capabilities. This standard relates to SCSI multipathing features and excludes multipathing between interconnect devices (such as Fibre Channel switches) and transport specific multipathing (such as iSCSI multiple connections per session).

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

The provisions of the referenced specifications other than ISO/IEC, IEC, ISO and ITU documents, as identified in this clause, are valid within the context of this International Standard. The reference to such a specification within this International Standard does not give it any further status within ISO or IEC. In particular, it does not give the referenced specification the status of an International Standard.

ISO/IEC 9899:1999, *Programming languages – C*

ISO/IEC 14165-133, *Information technology – Fibre channel – Part 133: Switch fabric-3 (FC-SW-3)*

ISO/IEC 14165-251, *Information technology – Fibre channel – Part 251: Framing and signalling (FC-FS)*

ISO/IEC 14776-115, *Information technology – Small computer system interface (SCSI) – Part 115: Parallel interface-5 (SPI-5)*

ISO/IEC 14776-150, *Information technology – Small computer system interface (SCSI) – Part 150: Serial attached SCSI (SAS)*

ISO/IEC 14776-413, *Information technology – Small computer system interface (SCSI) – Part 413: Architecture model-3 (SAM-3)*

ISO/IEC 14776-453, *Information technology – Small computer system interface (SCSI) – Part 453: Primary commands-3 (SPC-3)*

RFC 3720, *Internet Small Computer Systems Interface (iSCSI)*

NOTE Copies of IETF standards such as RFC 3720 may be obtained through the Internet Engineering Task Force (IETF) at http://www.ietf.org.

3 Terms, definitions and abbreviations

3.1 Terms and definitions

For the purposes of this document the following terms and definitions apply.

3.1.1
auto-failback
capability of some multipath drivers to resume use of a path when the path transitions from unavailable to available

3.1.2
auto-probing
capability of some multipath drivers to validate operational paths that are not currently being used

3.1.3
available paths
set of paths for a logical unit that may be considered for routing I/O requests

NOTE For symmetric access devices, all paths are considered *available*. For asymmetric access devices, all paths in active target port groups are considered *available*.

3.1.4
device file
operating system files (for instance UNIX, Linux etc.,) that facilitate communication with the system's hardware and peripherals

3.1.5
device identification VPD page
VPD page that provides the means to retrieve identification information about the SCSI device, logical unit, and SCSI port

3.1.6
hexadecimal-encoded binary data
ASCII character string used to denote the hexadecimal encoding of a binary string of octets

NOTE It may only contain the ASCII characters 0-9, A-F, and a-f. Two hexadecimal characters represent each byte of binary data.

3.1.7
host
compute node connected to the SAN

3.1.8
initiator
SCSI device that initiates requests; also known as a client

NOTE In this document, initiator refers to an initiator port.

3.1.9
logical unit
addressable entity within a SCSI target

NOTE For example, RAID arrays expose each virtual disk volume as a logical unit. When the term "logical unit: is used in this standard and is not qualified as a "multipath logical unit" or "path logical unit", it refers to a logical unit in a target device.

3.1.10
multipath logical unit
an object type of this API representing a "virtual "logical unit that coalesces multiple path logical units for the same underlying device logical unit

3.1.11
object ID
unique identifier assigned to any object within the MP API

NOTE Objects sometimes represent physical entities, e.g. initiator ports. At other times, objects represent logical entities, e.g. target port groups.

3.1.12
path
association between an initiator port, target port and logical unit, see 3.1.13

3.1.13
path logical unit
an object type of this API providing access to a single logical unit through a single initiator port and single device port

NOTE Within this API, each path (see 3.1.12) is modelled as a path logical unit. The result of multipath drivers is a single OS device file representing a multipath logical unit aggregating multiple path logical units.

3.1.14
persistent
quality of something being non-volatile

NOTE This usually means that the associated data is recorded on some non-volatile medium such as flash RAM or magnetic disk and that the data survives beyond system reboots. Implicitly, the data is readable from the non-volatile medium.

NOTE Examples of persistent storage:

- under Windows, the registry would be a common place to find persistently stored values (assuming that the values are not stored as volatile);
- under any OS a file on magnetic hard disk would be persistent.

3.1.15
plugin
software, written for an OS, HBA or device vendor that provides support for one or more multipath drivers

NOTE The plugin's job is to provide a bridge between the library's interface and the vendor's multipath driver. A plugin is typically a loadable module, for instance, a DLL in Windows and a shared object in UNIX. A plugin is accessed by an application through the Multipath Management API library.

3.1.16
product (or device product)
a particular model of target device, identified by the vendor, product and revision IDs returned in the standard SCSI INQUIRY command response

3.1.17
target
SCSI device containing logical units and SCSI target ports that receives commands from a SCSI initiator

3.1.18
target port group
set of target ports that are in the same target port access state at all times

3.1.19
unicode
system of uniquely identifying (numbering) characters such that nearly any character in any language is identified.

3.1.20
VPD
vendor specific information about a device returned in response to a SCSI INQUIRY command with the EVPD bit set (see SPC-3)

3.2 Abbreviations

API	application programming interface
DLL	dynamic link library
HBA	host bus adapter
LUN	logical unit number
OID	object identifier
OS	operating system
UML	Unified Modeling Language
UTF	Unicode Transformation Format
VPD	vital product data

4 Document conventions

The API is specified as a set of types and structures (see Clause 6) followed by a set of function definitions (see Clause 7). This clause discusses the formats used in these clauses along with conventions used in defining the API.

Constants are defined as a list of #defines followed by a typedef for a C integer type. C language enums do not have a specific size; using #defines rather than enums helps assure client code is interoperable across platforms and compilers – especially if used in C++ applications.

API description format

Each API's description is divided into seven subclauses.

1 **Synopsis**

This subclause gives a brief description of what action the API performs.

2 **Prototype**

This subclause gives a prototype of the function in a format that is a combination of a C function prototype and an Interface Definition Language (IDL) prototype. The prototypes show the following:

- the name of the API;
- the return type of the API;
- each of the parameters of the API, the type of each parameter, and whether that parameter is an input parameter, output parameter, or both an input and an output parameter.

3 **Parameters**

This subclause lists each parameter along with an explanation of what the parameter represents.

4 **Typical return values**

This subclause lists the Typical Return Values of the API with an explanation of why a particular return value would be returned. It is important to note that this list is not a comprehensive list of all of the possible return values. There are certain errors, e.g. MP_STATUS_INSUFFICIENT_MEMORY, which might be returned by any API.

5 **Remarks**

This subclause contains comments about the API that may be useful to the reader. In particular, this subclause contains extra information about the information returned by the API.

6 **Support**

This subclause states that if an API is mandatory to be supported, optional to be supported, or mandatory to be supported under certain conditions.

- If an API is mandatory to be supported a client can rely on the API functioning under all circumstances.
- If the API is optional to be supported then a client cannot rely on the API functioning.
- If the API is mandatory to be supported under certain conditions then a client can rely on the API functioning if the specified conditions are met. Otherwise a client should assume that the API is not supported.

7 **See also**

This subclause lists other related APIs or related code examples that the reader might find useful.

5 Background technical information

5.1 Overview

Open system platforms give applications access to physical devices by presenting a special set of file names that represent the devices. Although end users typically don't use these special device files, knowledgeable applications (file systems, databases, backup software) operate on these device files and provide familiar user interfaces to storage. The device files have a hierarchical organization, either by using files and directories or by naming conventions.

This hierarchy of device files (sometimes called a device tree) provides an effective interface for simpler, desktop device configurations. Inside open systems kernels, the hierarchy is exploited to allow different drivers to operate on different parts of the device tree. When the OS discovers connected devices and builds the device tree, multiple paths to the same device may show up as separate device files in the device tree. Separate storage applications using device files that represent paths to the same device will overwrite each other's data.

As storage products (typically disk arrays) strove for better reliability and performance, they added multipath support. For OSes that lacked multipath support, the device and logical volume manager vendors provided multipath drivers. Device standards lacked standard interfaces for identifying multipath devices; so multipath drivers are often limited to specific device products. Recently standards have been defined and OS vendors have started integrating multipath support in their bundled drivers.

These drivers create special device files that represent multipath devices. Storage applications like file systems can use these multipath device files the same way they would use a single-path device file, but benefit from improved reliability and performance. In addition, the multipath drivers provide some management capabilities, for example, failover or load balancing, that only apply to multipath devices.

This standard focuses on devices accesses through SCSI commands. SCSI commands are sent to a target device by an initiator. The target may consist of multiple logical units. For example, a RAID array exposes virtual disk as separate logical units. A target device supporting multiple paths and attached hosts will nearly always have multiple ports. Each permutation of initiator port, target port and logical unit is commonly referred to as a path. With no multipath support in place, the OS would see each path as separate logical units. The function of multipath drivers is then to create a virtual multipath device that aggregates all these path logical units.

5.2 Target port groups

A logical unit may only be accessible through certain target ports. If the device supports asymmetric access (see 5.3.2), certain ports may be preferred for access (sometimes this is referred to as affinity. ISO/IEC 14776-453 (SPC-3) has introduced target port groups as a way for target devices to represent access characteristics for logical units. A target port group is a collection of ports. All the logical units associated with that target port group share the same access state (active/optimized, active/non-optimized, standby or unavailable).
Target port groups are abstract elements that may or may not equate to an element of the target system (such as a controller).
The concept of target port groups can be applied to all devices, even if they don't actually implement the SCSI standard interfaces. This API does not require an SPC-3-compliant array; it includes target port groups and uses the terminology of ISO/IEC 14776-453 (SPC-3) as a starting point, but is extended to reflect common vendor implementations.
In order to simplify tasks for client software, all plugins/drivers make it appear that the underlying hardware uses target port group interfaces. For example, consider an asymmetric array with two ports where each port is primary (optimized) for half the logical units. The plugin/driver would create four "virtual" target port groups; each logical unit would be part of two target port groups, one with optimized access state for its primary controller and one with non-optimized access state for the secondary controller. See Annex B for more details.

5.3 Relationship between target port groups in SCSI and in this API

5.3.1 General

This subclause describes the relationship between the interfaces defined in ISO/IEC 14776-453 (SPC-3) and this API related to target port groups.
The SCSI **Device Identification VPD page (i.e., page 83h)** and REPORT TARGET PORT GROUPS command allow initiators to discover the target port group configuration.

- **Device Identification VPD page** returns a list of identifiers. These include:
    - **relative target port identifier** – a two-byte value with a target-unique ID for the target port the INQUIRY is sent to. In this API, this is the relativePortID property in MP_TARGET_PORT_PROPERTIES;
    - **target port group identifier** – a two-byte value with a target-unique ID for the target port group. In this API, this is the tpgID property of MP_TARGET_PORT_GROUP_PROPERTIES.
- The REPORT TARGET PORT GROUPS command returns a list of target port groups, with access state, and the list of relative port Ids of target ports that comprise each target port group. The access state corresponds to this API's MP_ACCESS_STATE_TYPE and MP_TARGET_PORT_GROUP accessState property.

The SCSI SET TARGET PORT GROUPS command allows an initiator to set target port access state – which causes failover or failback. This API provides MP_SetTPGAccess as an interface to SET TARGET PORT GROUPS.
For a concrete example, Figure 1 depicts a RAID array with asymmetric access and two controllers. Each controller contains two ports that always share the same access state. The RAID configuration is set up with four logical units. Optimally each pair of logical units is accessed through the ports on different controllers. In case either controller fails, all four logical units can be accessed through the ports in the alternate controller.

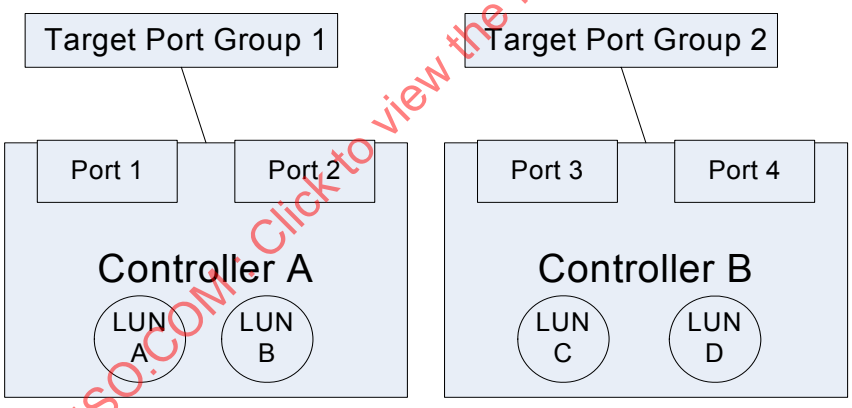

Figure 1 – Asymmetric array example

The table below summarizes the information returned for this array configuration in the SCSI INQUIRY identifiers and REPORT TARGET PORT GROUPS command response.

Logical unit	Access from port 1 or 2	Access from port 3 or 4
	TPG ID / State	TPG ID / State
A	1 / Active optimized	2 / Standby
B	1 / Active optimized	2 / Standby
C	1 / Standby	2 / Active optimized
D	1 / Standby	2 / Active optimized

In case of a failure condition of controller A, all logical units as accessed from port 1 or 2 will either see lack of response or a TPG access state of unavailable. Logical units A and B as seen through ports 3 or 4 will see an access state of active non-optimized.

Note that the target port group access states for a given target port group ID differs depending on which port the REPORT TARGET PORT GROUP command is issued to. In this API, each target port group ID and access state permutation is modelled as a different instance of a target port group class. The figure below is an instance diagram representing the API instances corresponding to this same asymmetric array described above. The relevant API properties are also included.

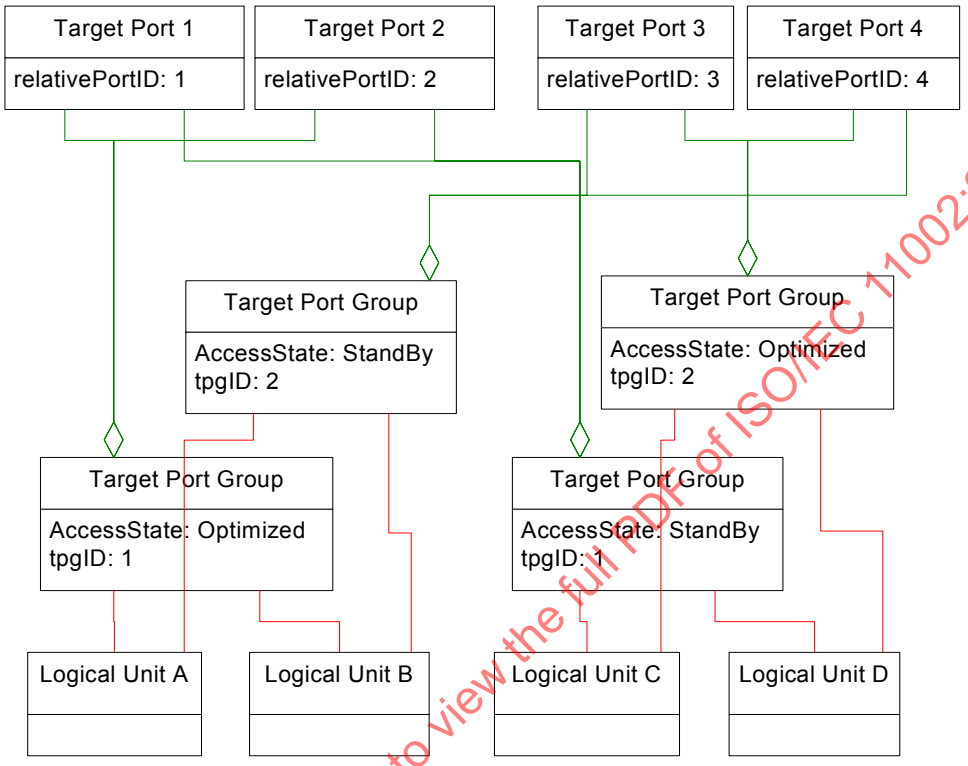

Figure 2 – API Instances corresponding to asymmetric array example

5.3.2 Symmetric and asymmetric multipath access

A multipath device may have symmetric or asymmetric access. There may be a performance cost when host drivers switch between asymmetric paths. Symmetric access devices avoid that penalty. One common asymmetric configuration is a RAID array where access to a particular logical unit is optimal through one device port and non-optimal through the other port. ISO/IEC 14776-453 (SPC-3) includes standard interfaces for discovery and management of multipath devices.[1] In addition to standardization of logical unit identifiers and a failover command, ISO/IEC 14776-453 (SPC-3) has interfaces that allow a target device to describe target port groups. All the ports in a target port group share an access state that is either optimal or non-optimal.

Setting the access state to active/optimized in all target ports groups associated with a logical unit indicates symmetric access. A target system where all logical units have symmetric access from all ports could be described with a single target port group with access state active/optimized associated with all logical units and target ports.

A logical unit could have symmetric access through some, but not all ports. The optimal ports can be used for load balancing, but the non-optimal ports should only be used for failover. This would be modelled with target port groups with multiple associated ports and access state set to active/optimized.

---

1 Although this API provides interfaces for discovery of multipath devices, it only provides information available through installed plugins. If a client application requires comprehensive discovery of all devices, it should also use platform-specific device discovery APIs.

5.3.3 Logical unit affinity groups

Some target devices (particularly RAID arrays) have groups of logical units that failover/failback as a group. In other words, when one logical units' target port group access state changes, the access state of the other logical units in the group also changes. ISO/IEC 14776-453 (SPC-3) has a simple interface to discover these groups; the **Device Identification VPD page response** may include a logical unit group identifier (identifier type 6h). All the logical units that expose the same logical unit group identifier are members of the same logical unit group. A logical unit may only be a member of a single logical unit group.
This API follows the same approach; MP_MULTIPATH_LOGICAL_UNIT_PROPERTIES has a property logicalUnitGroupID. The details for this property (see 6.26) specify how a plugin/driver sets this property if the target device does not support the SCSI interface.
This API does not provide a mechanism to create a logical unit group or add members. ISO/IEC 14776-453 (SPC-3) does not provide this capability. In some implementations, the logical unit groups are artefacts of other target capabilities. For example, the logical unit groups in some arrays follow the RAID topology of the configuration of snapshots. Due to the overlap with these other target features, no interfaces for modification are provided in this API.

5.3.4 Load balancing

5.3.4.1 General

This API includes four interfaces that influence load balancing.

- When multiple paths are available with the same access state, each individual I/O request can only be issued to one specific path. Multipath drivers may allow the administrator to select an **algorithm** used to determine which path is selected.
- Some drivers allow the administrator to select a subset of available paths as most preferred; assuming no errors are encountered, I/Os are restricted to the preferred paths. But the non-preferred paths are not necessarily taken off-line; if the preferred paths become not-available, the non-preferred paths may be used as a fallback. This capability is implemented entirely in the host drivers and is independent of target port groups. Some drivers allow multiple levels of preferences, referred to as **administrative weights** in this API.
- Drivers may also allow an administrator to specify an **override path,** a path that is temporarily used for all I/O.
- Drivers may also allow an administrator to **disable a path,** make a path temporarily unavailable for load balancing.

The subclauses below describe these interfaces in more detail.

5.3.4.2 Load balancing algorithms

The API allows a plugin/driver to advertise multiple load balance algorithms that an API client can offer to the administrator. Several common algorithms are defined in MP_LOAD_BALANCE_TYPE. The plugin/driver can extend this list with driver-specific algorithms. The API treats these proprietary algorithms opaquely, but provides a mechanism for the plugin/driver vendor to expose a vendor and algorithm name to client applications. A client could use these names to populate a "pull-down" list of load balance algorithms that includes vendor-specific algorithms.

Some multipath drivers have load-balancing algorithms optimized for certain device types. The device type is determined by the vendor and product IDs returned in the SCSI Inquiry data. A plugin/driver can report its list of supported device types using MP_DEVICE_PRODUCT_PROPERTIES.

5.3.4.3 Administrative preference – path weight

Path weight is a value assigned by an administrator specifying a preference to assign to a path (or path logical unit). The drivers will actively use all available paths with the highest weight (see below for clarification of available). This allows an administrator to assign a subset of available paths for load balanced access and reserve the others as backup paths. For symmetric access devices, all paths are considered available. For asymmetric access devices, all paths in active target port groups are considered available.

The range of weights (maximumWeight) supported by the driver is exposed to clients as a plugin/driver property. A driver with no path weight capabilities should set this property to zero. A driver with the ability to enable/disable paths should set this property to 1. Plugins/drivers with more weight settings can set the property appropriately.

Path weight has precedence over driver policy regarding path selection. In other words, if the drivers understand that a path with a lower weight may be optimal, they should still limit routing to paths the administrator has assigned the highest weight.

Other APIs may impact I/O routing (MP_DisablePath, MP_EnablePath, MP_SetOverridePath, MP_SetTPGAccess) but no other API changes the actual weight values. This approach allows an administrator to define long-term policy using path weights and temporarily to override this policy in order to address hardware failures, run diagnostic tests or quiesce hardware.

The default weight (prior to being set by the administrator) is the plugin/driver's maximumWeight value.

Path weight shall be persisted by the plugin/driver.

Example:

A host has four paths to a LUN on a device with asymmetric access; in the normal case, paths one and two are active and paths three and four are in standby state. The administrator would prefer that:

- during non-failover periods, I/O should be through path 1;
- if an HBA failure impacts path 1, but the device is not in a failover state, then path 2 should be used; and
- if the device is in failover state (making paths 1 and 2 unusable) and all HBAs are functioning, then path three should be used.

To configure these preferences, the administrator would assign weight 2 to paths one and three and weight 1 to paths two and four. Actually, the value of the weights is not important as long as the weights assigned to paths one and three are higher than those assigned to paths two and four, respectively.

5.3.4.4 Disable load balancing – override path

The plugin/driver may optionally provide an interface (MP_SetOverridePath) for an override path. An override path is a single path that the administrator can specify for all I/O to a logical unit. Setting a preferred path will disable load balancing. Path weights are not changed when a path is overridden.

5.3.4.5 Disable path

The plugin/driver may optionally provide an interface (MP_DisablePath) to disable a path. Disabling a path makes it ineligible for load balancing in the future, but it may stay in use while the drivers migrate activity to a different path. Path weights are not changed when a path is disabled.

5.3.5 Model overview

The model for this API contains the following classes:

- **Library** – the client library interface that front ends all the plugins;
- **Device product** – information about a specific device supported by the driver;
- **Plugin** – the driver-specific library implementing this API;
- **Proprietary load balance types** – vendor name and description for driver-specific load balance algorithms; opaque to the API, provides algorithm names to applications;
- **Initiator port** – a port on the system hosting the plugin;
- **Target port** – a port on the device;
- **Path logical unit** – represents a single initiator/target port combination accessing a logical unit. May not have a corresponding OS device file name;
- **Multipath logical unit** – the virtual device the aggregates all paths (path logical units) referencing the same logical unit; and
- **Target port group** –a set of target ports that share a common access state.

Figure 3 is a UML diagram that shows the relationship between the various classes of objects in the Multipath model.

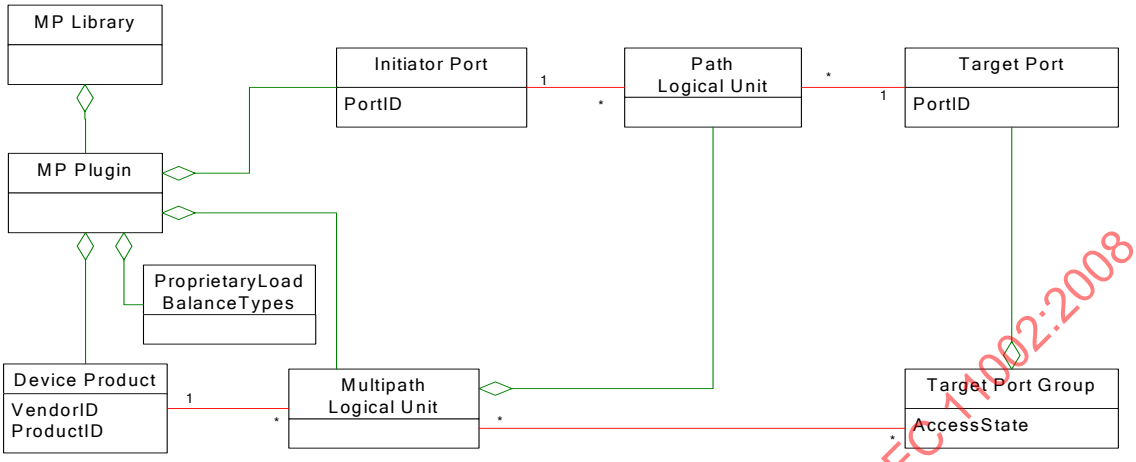

Figure 3 – Relationship between various objects in the multipath model

The structures and APIs defined below allow a client to navigate this model in order to discover and manipulate multipath drivers and hardware. Each class in the diagram has a structure containing properties (for example, MP_INITIATOR_PORT_PROPERTIES has properties for an initiator port) and an API to get the properties (MP_GetInitiatorPortProperties). Other APIs exist to allow the client to follow the associations in the diagram above. For example, the rightmost vertical line represents an aggregation of target ports in a target port group; MP_GetTargetPortOidList returns a list of target port OIDs (OIDs act something like pointers, see 5.6.3). Other APIs change behavior by setting specific properties or by operating on groups of objects.

Figure 4 below is a UML instance diagram that depicts the OS/driver view of a configuration with four paths connected to the same logical unit (for example, a RAID volume). Two initiator ports are connected to separate pairs of target ports, one optimized and one non-optimized, for the particular logical unit. The model depicts the typical MP driver behavior of treating the multipath logical unit as an aggregation of non-MP device files rather than an aggregation of paths.

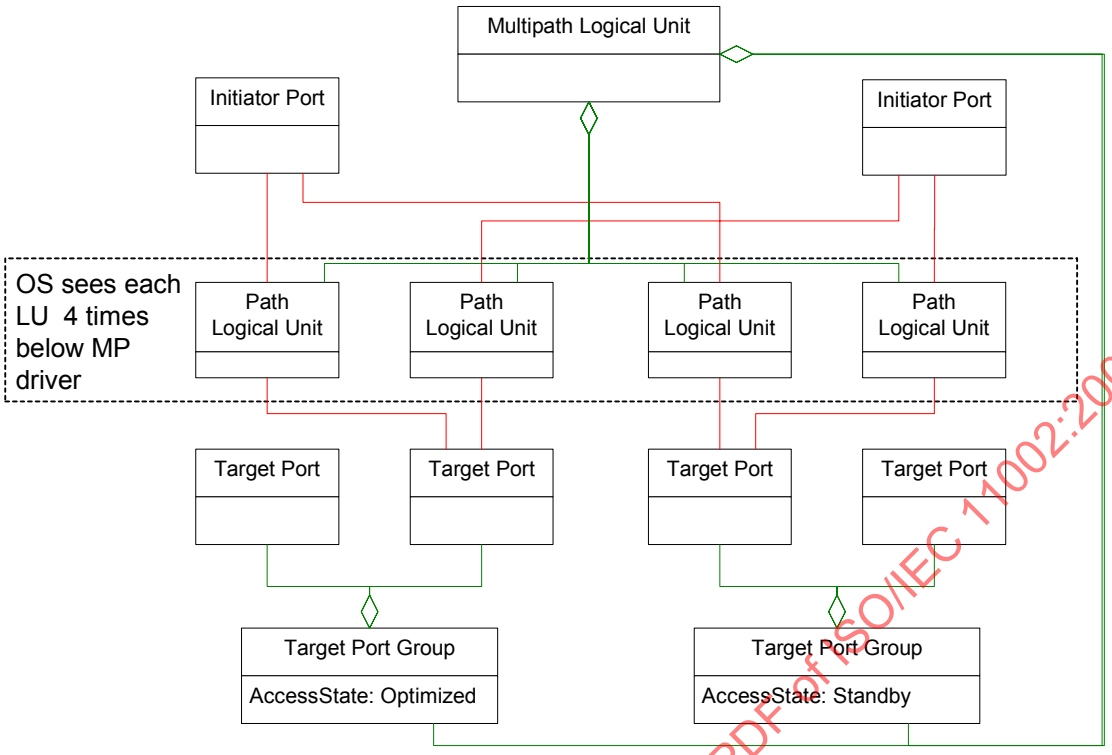

Figure 4 – Driver representation of a logical unit with multiple paths

Note that class/structure instances are not shared across plugin/drivers. But instances in separate plugin/drivers may map to the same "real world" object. For example, multiple plugin/drivers may represent the same initiator (HBA) port. A client would determine these ports are the same by comparing the port name (for example, FC Port WWN) properties of the port instances from the different plugin/drivers.

Installation and configuration of multipath drivers can be complex and hazardous. In some cases, overlap between plugin/drivers could represent configurations that may be catastrophic for a customer. This API does not enforce "best practices". It assumes that the customer has installed drivers in a "safe" manner. This API just reports on, and manipulates, the configuration.

5.4 Client discovery of optional behavior

5.4.1 General

Without multipath drivers, it's usually straightforward to get a list of all the disks attached to a system; usually this is just a list of all the device files with names indicating they are disks. But with MP drivers installed, it may be difficult to determine which device files are subsumed by a virtual multipath device. And the multipath driver may add additional special names to the list of disk devices. The primary objective of this API is to create a deterministic way for management software to discover the storage resources attached to a server.

In addition to the discovery functions, this API also provides functions for active management of multipath drivers, functions to control failover/failback and load balancing. These active management APIs are optional.

In general, support for optional behavior is exposed through properties of plugin/drivers (and other objects). For example, MP_PLUGIN_PROPERTIES has a property canActivateTPGs that informs a client whether this plugin supports failover/failback commands.

5.4.2 Discovery of load balancing behavior

This API has built-in support for common load balancing algorithms, but also allows plugins to describe proprietary algorithms. These are simply exposed as opaque information that a client can display or modify and are not actually interpreted by the API.

The client can determine the available load balance algorithms by looking at the supportedLoadBalanceTypes property of MP_PLUGIN_PROPERTIES returned by MP_GetPluginProperties. If MP_LOAD_BALANCE_TYPE_PRODUCT is set in supportedLoadBalanceTypes, then the client should also use MP_GetDeviceProductOidList and MP_GetDeviceProductProperties to get a list of target product types supported by the plugin. If there is an MP_DEVICE_PRODUCT_PROPERTIES instance with the same vendor, product and revision IDs as a specific logical unit, then the supportedLoadBalanceTypes property in that MP_DEVICE_PRODUCT_PROPERTIES instance override the plugin-wide supportedLoadBalanceTypes.

The client can determine the current load balance algorithm for a specific logical unit by looking at the currentLoadBalanceType property of MP_MULTIPATH_LOGICAL_UNIT_PROPERTIES returned by MP_GetMPLogicalUnitProperties.

The client can set the load balance algorithm for a specific logical unit using MP_GetMPLogicalUnitProperties and specifying a value other than MP_LOAD_BALANCE_TYPE_UNKNOWN for currentLoadBalanceType. MP_LOAD_BALANCE_TYPE_PRODUCT is only valid if vendor, product and revision from MP_GetMPLogicalUnitProperties match those in an instance of MP_DEVICE_PRODUCT_PROPERTIES returned by MP_GetDeviceProductProperties.

The client can set a plugin-wide default using MP_SetPluginLoadBalanceType.

For example, imagine an MP driver from Yoyodyne Corporation supports the following load balancing algorithms:

- round robin (the default);
- least IO;
- two algorithms created by the driver-writers for any device types (known as YY1 and YY2);
- an algorithm for one particular array model (the Acme 3500 array); and
- the YY1 algorithm is not supported for Acme 3500 logical units.

The driver does not allow the administrator to set different load balance types for different logical units on a target.

There should be 1 instance of MP_PLUGIN_PROPERTIES with the following flags set in supportedLoadBalanceTypes:

MP_LOAD_BALANCE_ROUNDROBIN	2	2h
MP_LOAD_BALANCE_TYPE_LEASTIO	8	8h
MP_LOAD_BALANCE_TYPE_PRODUCT	16	10h
MP_LOAD_BALANCE_TYPE_PROPRIETARY1	65536	10000h
MP_LOAD_BALANCE_TYPE_PROPRIETARY2	131072	20000h

The value of supportedLoadBalanceTypes of MP_PLUGIN_PROPERTIES in hex would be 3001ah (the sum of these load balance type flags).

The value of defaultLoadBalanceType in MP_PLUGIN_PROPERTIES would be MP_LOAD_BALANCE_ROUNDROBIN.

There will be an instance of MP_PROPRIETARY_LOAD_BALANCE_PROPERTIES for each flag 65536 and up. This object has three fields, the index from above, a name for the algorithm and a name for the vendor. So in this example, we'll have of these objects:

- 65536, "YY1", "Yoyodyne Corp."
- 131072, "YY2", "Yoyodyne Corp."

Since MP_LOAD_BALANCE_TYPE_PRODUCT is set, there will also be an instance of MP_DEVICE_PRODUCT_PROPERTIES for each device with special driver load support. In this example, there will be one instance with vendor set to "ACME ", product set to "3500 ", and revision set to four nulls (this driver supports all revisions of the ACME 3500). The supportedLoadBalanceTypes for Acme 3500 will be set to 2001ah – the same as the plugin-wide supportedLoadBalanceTypes but without the bit for the YY1 algorithm.

Any logical unit on an Acme 3500 array can have currentLoadBalanceType in MP_MULTIPATH_LOGICAL_UNIT_PROPERTIES set to any of the four load balance types in the table above.

Any logical unit of non-Acme-3500 targets can have currentLoadBalanceType set to any of these load balance types other than MP_LOAD_BALANCE_TYPE_PRODUCT.

5.4.3 Client discovery of failover/failback capabilities

Failover only applies to asymmetric access devices. A client can discover whether a logical unit is on an asymmetric access device by looking at the MP_MULTIPATH_-LOGICAL_UNIT.asymmetric property.
MP_MULTIPATH_LOGICAL_UNIT. canActivateTPGs indicates support for the MP_Activate-TPGs API – this API provides manual failover capabilities.

5.4.4 Client discovery of a driver's OS device file name behavior

Some multipath drivers leave the underlying OS device file names (those representing path logical units) on this system. This behavior can be tested with MP_PLUGIN_-PROPERTIES.exposesPathDeviceFiles. If exposesPathDeviceFiles is set to false, then the plugin will only expose a single device file name for a multipath logical unit.
If MP_PLUGIN_PROPERTIES.exposesPathDeviceFiles is true, then multiple device file names are available for a multipath logical unit, one for each path.
Some multipath drivers create OS Device Files in non-standard locations. This behavior can be tested with MP_PLUGIN_PROPERTIES.deviceFileNamespace. If this property is null, the device file names associated with the plugin/driver match the "usual" platform names as documented in A.3. If deviceFileNamespace is non-null it is a simple regular expression describing the format for device file names, documented in the deviceFileNamespace property of MP_PLUGIN_PROPERTIES (see 6.24).

5.4.5 Client discovery of auto-failback capabilities

Auto-failback is a capability of some multipath drivers to resume use of a path when the path transitions from unavailable to available. In some cases, this is accomplished with polling (the driver attempts IOs on unavailable paths).
MP_PLUGIN_PROPERTIES.autoFailbackSupport describes the driver's support for auto-failback. MP_AUTOFAILBACK_SUPPORT_PLUGIN indicates auto-failback is managed the same across all devices. MP_AUTOFAILBACK_SUPPORT_MPLU indicates auto-failback settings are set separately for each multipath logical unit. MP_AUTOFAIL-BACK_SUPPORT_PLUGINANDMPLU indicates that the both global and per-multipath logical unit settings are supported.
If autoFailbackSupport is either MP_AUTOFAILBACK_SUPPORT_PLUGIN or MP_AUTOFAIL-BACK_SUPPORT_PLUGINANDMPLU, then these plugin properties are defined:

pluginAutofailbackEnabled
: True if the administrator has requested that auto-failback be enabled for all paths accessible via this plugin

failbackPollingRateMax
: The maximum polling rate (in seconds) supported by the driver. Zero (0) indicates the driver either does not poll for auto-failback or has not provided an interface to set the polling rate.

currentFailbackPollingRate
: The current polling rate (in seconds) for auto-failback. This cannot exceed failbackPollingRateMax.

If autoFailbackSupport is either MP_AUTOFAILBACK_SUPPORT_MPLU or MP_AUTOFAIL-BACK_SUPPORT_PLUGINANDMPLU, then these multipath logical unit (MP_MULTIPATH_-LOGICAL_UNIT_PROPERTIES) properties are defined:

autofailbackEnabled
: MP_TRUE if the administrator has requested that auto-failback be enabled for this multipath logical unit. If the plugin's autoFailbackSupport is MP_AUTOFAIL-BACK_SUPPORT_PLUGINANDMPLU, MP_UNKNOWN is valid and indicates that multipath logical unit has auto-failback enabled if pluginAutofailbackEnabled is true.

failbackPollingRateMax
: The maximum polling rate (in seconds) supported by the driver. Zero (0) indicates the driver either does not poll for auto-failback or has not provided an interface to set the polling rate for multipath logical units. If this property and the plugin's

failbackPollingRateMax are non-zero, this value has precedence for the associate logical unit.

currentFailbackPollingRate
The current polling rate (in seconds) for auto-failback. This cannot exceed failbackPollingRateMax. If this property and the plugin's currentFailbackPollingRate are non-zero, this value has precedence for the associate logical unit.

5.4.6 Client discovery of auto-probing capabilities

Auto-probing is an optional capability to validate operational paths that are not currently being used.

MP_PLUGIN_PROPERTIES.autoProbingSupport describes the driver's support for auto-probing. MP_AUTOPROBING_SUPPORT_PLUGIN indicates auto-probing is managed the same across all devices. MP_AUTOPROBING_SUPPORT_MPLU indicates auto-probing settings are set separately for each multipath logical unit. MP_AUTO-PROBING_SUPPORT_PLUGINANDMPLU indicates that the both global and per-multipath logical unit settings are supported.

If autoProbingSupport is either MP_AUTOPROBING_SUPPORT_PLUGIN or MP_AUTO-PROBING_SUPPORT_PLUGINANDMPLU, then these plugin properties are defined:

pluginAutoProbingEnabled
True if the administrator has requested that auto-probing be enabled for all paths accessible via this plugin

probingPollingRateMax
The maximum polling rate (in seconds) supported by the driver. Zero (0) indicates the driver either does not poll for auto-probing or has not provided an interface to set the polling rate.

currentProbingPollingRate
The current polling rate (in seconds) for auto-probing. This cannot exceed probingPollingRateMax.

If autoProbingSupport is either MP_AUTOPROBING_SUPPORT_MPLU or MP_AUTO-PROBING_SUPPORT_PLUGINANDMPLU, then these multipath logical unit properties are defined:

autoProbingEnabled
MP_TRUE if the administrator has requested that auto-probing be enabled for this multipath logical unit. If the plugin's autoProbingSupport is MP_AUTO-PROBING_SUPPORT_PLUGINANDMPLU, MP_UNKNOWN is valid and indicates that multipath logical unit has auto-probing enabled if pluginAutoProbingEnabled is true.

probingPollingRateMax
The maximum polling rate (in seconds) supported by the driver. Zero (0) indicates the driver either does not poll for auto-probing or has not provided an interface to set the polling rate for multipath logical units. If this property and the plugin's probingPollingRateMax are non-zero, this value has precedence for the associate logical unit.

currentProbingPollingRate
The current polling rate (in seconds) for auto-probing. This cannot exceed probingPollingRateMax. If this property and the plugin's currentFailbackPollingRate are non-zero, this value has precedence for the associate logical unit.

5.4.7 Client discovery of support for LU assignment to target port groups

If an asymmetric access device allows logical units to be assigned to target port groups, MP_TARGET_PORT_GROUP.supportsLuAssignment will be true. This indicates that MP_AssignLogicalUnitToTPG API is available.

5.5 Events

A long-running application may subscribe to events and be asynchronously notified of changes. The API has two types of events:

- visibility changes – when objects appear or disappear; and
- property changes – when properties in an object change.

APIs allow clients to register or deregister for each type of event. Registration specifies the address of a client-supplied callback method that is invoked when events occur. The client can specify a specific object type (defaults to all object types). The client can specify a

specific plugin (defaults to all plugins). Multiple calls allow registration for a subset of object types and plugins.

The client can also specify “caller data” that may be used by the caller to correlate the event to source of the registration. The plugin saves the caller data and returns it with each event.

See these subclauses for more detail about events: 6.10, 6.11, 7.5, 7.6, 7.35 and 7.36.

5.6 API concepts

5.6.1 Library and plugins

A multipath management API implementation that is compliant with this standard shall facilitate common management methodologies for configurations of multipath implementations provided by multiple vendors and installed or removed at various times. This dynamic installation and removal shall be achieved by a software structure that is either OS specific and fully documented by the OS vendor or as defined in this standard for an OS independent structure (see 5.6.2).

Functions that are shown as mandatory but are not relevant in an OS specific implementation shall return MP_STATUS_SUCCESS (e.g., MP_DeregisterPlugin) so that applications will not have to code to the specific underlying implementation.

5.6.2 OS-independent implementation

The Multipath Management API may be implemented using a combination of a library and plugins.
Among other things, the library is responsible for loading plugins and dispatching requests from a management application to the appropriate plugin(s).
In an OS-independent implementation, OS, HBA or device vendors provide plugins to manage subsets of target devices. Typically, a plugin will take a request in the generic format provided by the library and then translate that request into a vendor specific format and forward the request onto the vendor’s device driver. In practice, a plugin may use a DLL or shared object library to communicate with the device driver. Also, it may communicate with multiple device drivers. Ultimately, the method a plugin uses to accomplish its work is entirely vendor specific.

Although rare, two plugins may model the same real-world resource. This could apply to initiator or target ports or even logical units. The client determines equivalence by testing the properties that contains names/Ids reported by the hardware itself (such as Port WWNs for FC ports). If the client application is operating across multiple hosts, the same approach is used to look for occurrences of the same target port of logical unit connected to multiple hosts. This allows a client to have a single instance that aggregates information from several plugins. One consequence of this overlap is that multiple plugins may report the same event to the client.

This architecture has no boot-time requirements. Plugins are registered with the common library when they are installed. This would typically be done when MP drivers (and/or management clients) are installed on the system. The registration information is persistent and resides in either a registry or in a configuration file (see 7.37).

5.6.3 Object ID

The core element of the Multipath Management API is the object ID (OID). An object ID is a structure that “uniquely” identifies an object. The reason uniquely is in quotes in the previous sentence is that it is possible, though very unlikely, that an object ID would be reused and refer to a different object.

An object ID consists of three fields:

a) An object type. This identifies the type of object, e.g. port, logical unit, etc., that the object ID refers to;

b) An object owner identifier. This is a number that is used to uniquely identify the owner of the object. Either the library or a plugin owns objects; and

c) An object sequence number. This is a number used by the owner of an object, possibly in combination with the object type, to identify an object.

The combination of these properties assures that object Ids are unique across plugins.

To a client that uses the library, object IDs shall be considered opaque. A client shall use only documented APIs to access information found in the object ID.

There are several rules for object IDs that the library, plugins and clients shall follow. They are:

an object ID can only refer to one object at a time;

an object can only have one object ID that refers to it at any one time. It is not permissible to have two or more object IDs that refer to the same object at the same time. In some cases this may be difficult, but the rule still shall be followed.

For example, suppose a HBA port is in a system. That HBA port will have an object ID. If the HBA is removed and then reinserted (while the associated plugin is running) then one of two things can happen;

a) the HBA port can retain the same object ID as it had before it was removed; or

b) The HBA port can get a new object ID and the old object ID will no longer be usable.

This can only happen if the same HBA is reinserted. If a HBA is removed and another HBA is inserted that has not been in the system while a particular instance of the library and plugins are running then that HBA port shall be given a new object ID.

The library and plugins can uniquely identify an object within their own object space by using either the object sequence number or by using the object sequence number in combination with the object type. Which method is used is up to the implementer of the library or plugin.

Object sequence numbers shall be reused in a conservative fashion to minimize the possibility that (due to wrapping of the sequence number) an object ID will ever refer to two (or more) different objects in any one instance of the library or plugin. This rule for reuse only applies to a particular instance of the library or plugin. Neither the library nor plugins are required or expected to persist object sequence numbers across instances.

Because neither the library nor plugins are required to persist object sequence numbers a client using the library shall not use persisted object IDs across instances of itself.

Similarly, different instances of the library and plugins may use different object IDs to represent the same physical entity.

5.6.4 Object ID list

An object ID list is a list of zero or more object IDs. There are several APIs, e.g., MP_GetTargetPortOidList, that return object ID lists. Once a client is finished using an object ID list the client shall free the memory used by the list by calling the MP_FreeOidList API.

6 Constants and structures

6.1 MP_WCHAR

Typedef'd as a wchar_t (wchar_t is part of the ISO C programming standard ISO/IEC 9899:1999) and is available is all recent C compilers, though you may need special options to enable it).

6.2 MP_CHAR

Typedef'd as a char. Only used in contexts where wide characters cannot be used, such as filenames and ASCII text returned from SCSI commands.

6.3 MP_BYTE

An 8-bit unsigned value. Typedef'd as an unsigned char.

6.4 MP_BOOL

Typedef'd to an MP_UINT32. A variable of this type can have either of the following values:

- MP_TRUE
  This symbol has the value 1.

- MP_FALSE
  This symbol has the value 0.

6.5 MP_XBOOL

Typedef'd to an MP_UINT32. This is an extended boolean. A variable of this type can have any of the following values:

- MP_TRUE
  This symbol has the value 1.

- MP_FALSE
  This symbol has the value 0.

- MP_UNKNOWN
  This symbol has the value FFFFFFFFh.

6.6 MP_UINT32

A 32-bit unsigned integer value.

6.7 MP_UINT64

A 64-bit unsigned integer value.

6.8 MP_STATUS

Status values

MP_STATUS_SUCCESS
: This status value is returned when the requested operation is successfully carried out. This symbol has a value of 0.

MP_STATUS_INVALID_PARAMETER
: This status value is returned when parameter(s) passed to an API is detected to be invalid or inappropriate for a particular API parameter. If the parameter is an object ID, this status indicates that the object type subfield is defined in this standard, but is not appropriate for this API. This symbol has a value of 1.

MP_STATUS_UNKNOWN_FN
: This status value is returned when a client function passed into the API is not a previously registered/known function. This symbol has a value of 2.

MP_STATUS_FAILED
: This status value is returned when the requested operation could not be carried out. This symbol has a value of 3.

MP_STATUS_INSUFFICIENT_MEMORY
: This status value is returned when the API could allocate the memory required to complete the requested operation. This symbol has a value of 4.

MP_STATUS_INVALID_OBJECT_TYPE
: This status value is returned when an object id includes a type subfield that is not defined in this standard. This symbol has a value of 5.

MP_STATUS_OBJECT_NOT_FOUND
: This status value is returned when the object associated with the id specified in the API could not be located or has been deleted. Note that an invalid object type is covered by MP_STATUS_INVALID_OBJECT_TYPE so this status is limited to invalid object owner identifier or sequence number. This symbol has a value of 6.

MP_STATUS_UNSUPPORTED
: This status value is returned when the implementation does not support the requested function. This symbol has a value of 7.

MP_STATUS_FN_REPLACED
: This status value is returned when a client function passed into the API replaces a previously registered function. This symbol has a value of 8.

MP_STATUS_ACCESS_STATE_INVALID
This status value is returned when a device processing MP_SetTPGAccess returns a status indicating the caller is attempting to establish an illegal combination of access states. This symbol has a value of 9.

MP_STATUS_PATH_NONOPERATIONAL
This status is returned when communication cannot be established with the path selected by the caller. This symbol has a value of 10.

MP_STATUS_TRY_AGAIN
This status is returned when the plugin/driver is unable to complete the request at this time, but may be able to complete it later. This symbol has a value of 11.

MP_STATUS_NOT_PERMITTED
The operation is not permitted in the current configuration, but may be permitted in other configurations. This symbol has a value of 12.

6.9 MP_PATH_STATE

MP_PATH_STATE is an enumeration used to indicate the status of a path. This status is not returned by APIs, but is included in MP_PATH_LOGICAL_UNIT_PROPERTIES along with other path properties.

Constants

```
#define MP_PATH_STATE_OKAY                  0
#define MP_PATH_STATE_PATH_ERR              1
#define MP_PATH_STATE_LU_ERR                2
#define MP_PATH_STATE_RESERVED              3
#define MP_PATH_STATE_REMOVED               4
#define MP_PATH_STATE_TRANSITIONING         5
#define MP_PATH_STATE_OPERATIONAL_CLOSED    6
#define MP_PATH_STATE_INVALID_CLOSED        7
#define MP_PATH_STATE_OFFLINE_CLOSED        8
#define MP_PATH_STATE_UNKNOWN               9

typedef MP_UINT32 MP_PATH_STATE;
```

Definitions

MP_PATH_STATE_OKAY
The path is okay.

MP_PATH_STATE_PATH_ERR
The path is unusable due to an error on this path and no SCSI status was received.

MP_PATH_STATE_LU_ERR
A SCSI status was received for an I/O through this path indicating an error on the logical unit.

MP_PATH_STATE_RESERVED
The path is unusable due to a SCSI reservation.

MP_PATH_STATE_REMOVED
The path is not used because the OS or other drivers marked the path unusable.

MP_PATH_TRANSIITIONING
The path is transitioning between two valid states.

MP_PATH_STATE_OPERATIONAL_CLOSED
The path appears operational, but has not been opened. This state only applies to platforms that allow paths to be opened or closed.

MP_PATH_STATE_INVALID_CLOSED
No open was attempted but background probing determined that the path was dead.

MP_PATH_STATE_OFFLINE_CLOSED
The path appears operational, but has not been opened.

MP_PATH_STATE_UNKNOWN
The path is not operational, but the exact cause is not known.

Remarks

The error states are generally discovered when an I/O requests do not complete with normal status. The I/O request involved in this state change may have been issued by the multipath plugin/driver or by a user application. This standard does not require that the

plugin/driver poll for error conditions. If these error states are known, they may be returned; if details are not known, MP_PATH_STATE_UNKNOWN should be returned.

6.10 MP_OBJECT_VISIBILITY_FN

Format

```
typedef void (* MP_OBJECT_VISIBILITY_FN)(
   /* in */ MP_BOOL    becomingVisible,
   /* in */ MP_OID_LIST *pOidList,
   /* in */ void        *pCallerData
);
```

Parameters

becomingVisible

An MP_BOOL value indicating that the list of object specified by *pOidList* have become visible or have disappeared. A value of MP_TRUE indicates the objects have become visible. A value of MP_FALSE indicates the objects have disappeared.

pOidList

A list of IDs of objects whose visibility is being changed. All objects referenced shall be of the same type (different types may have different pCallerData values). All objects referenced shall all have become visible or have disappeared.

pCallerData

The pCallerData passed into MP_RegisterForObjectVisibilityChanges. This may be used by the caller to correlate the event to source of the registration.

Remarks

This type is used to declare client functions that can be used with the MP_RegisterForObjectVisibilityChanges and MP_DeregisterForObjectVisibilityChanges APIs.

When the client function is finished using the list referenced by pOidList, it shall free the memory used by the list by calling MP_FreeOidList.

6.11 MP_OBJECT_PROPERTY_FN

Format

```
typedef void (* MP_OBJECT_PROPERTY_FN)(
   /* in */ MP_OID_LIST *pOidList,
   /* in */ void        *pCallerData
);
```

Parameters

pOidList

A list of IDs of objects whose property values are being changed. All objects referenced shall be of the same type (different types may have different pCallerData values)

pCallerData

The pCallerData passed into MP_RegisterForObjectPropertyChanges. This may be used by the caller to correlate the event to source of the registration.

Remarks

This type is used to declare client functions that can be used with the MP_RegisterForObjectPropertyChanges and MP_DeregisterForObjectPropertyChanges APIs.

When the client function is finished using the list referenced by pOidList, it shall free the memory used by the list by calling MP_FreeOidList.

6.12 MP_OBJECT_TYPE

MP_OBJECT_TYPE is an enumeration used to differentiate API objects that are referenced by object Ids (odes). MP_OBJECT_TYPE is not directly used by clients, but is used to form object Ids.

Constants

```
#define MP_OBJECT_TYPE_UNKNOWN             0
#define MP_OBJECT_TYPE_PLUGIN              1
#define MP_OBJECT_TYPE_INITIATOR_PORT      2
#define MP_OBJECT_TYPE_TARGET_PORT         3
#define MP_OBJECT_TYPE_MULTIPATH_LU        4
#define MP_OBJECT_TYPE_PATH_LU             5
#define MP_OBJECT_TYPE_DEVICE_PRODUCT      6
#define MP_OBJECT_TYPE_TARGET_PORT_GROUP   7
#define MP_OBJECT_TYPE_PROPRIETARY_LOAD_BALANCE 8

typedef MP_UINT32 MP_OBJECT_TYPE;
```

Definitions

MP_OBJECT_TYPE_UNKNOWN
: The object has an unknown type. If an object has this type its most likely an uninitialized object.

MP_OBJECT_TYPE_PLUGIN
: Object type to identify a plugin module.

MP_OBJECT_TYPE_INITIATOR_PORT
: Object type to identify an initiator port.

MP_OBJECT_TYPE_TARGET_PORT
: Object type to identify an initiator port.

MP_OBJECT_TYPE_MULTIPATH_LU
: Object type to identify the multipath logical unit.

MP_OBJECT_TYPE_PATH_LU
: Object type to identify the path logical unit.

MP_OBJECT_TYPE_DEVICE_PRODUCT
: Object type to identify the device product.

MP_OBJECT_TYPE_TARGET_PORT_GROUP
: Object type to identify the target port group.

MP_OBJECT_TYPE_PROPRIETARY_LOAD_BALANCE
: Object type to identify a proprietary load balance type.

6.13 MP_OID

Format

```
typedef struct _MP_OID
{
    MP_OBJECT_TYPE   objectType;
    MP_UINT32        ownerId;
    MP_UINT64        objectSequenceNumber;
} MP_OID;
```

Fields

objectType
: Specifies the type of object. When an object ID is supplied as a parameter to an API the library uses this value to ensure that the supplied object's type is appropriate for the API that was called.

ownerId
: A number determined by the library that it uses to uniquely identify the owner of an object. The owner of an object is either the library itself or a plugin. When an object ID is supplied as a parameter to an API the library uses this value to determine if it should handle the call itself or direct the call to one or more plugins.

objectSequenceNumber
A number determined by the owner of an object, that is used by the owner possibly in combination with the object type, to uniquely identify an object.

Remarks

Clients of the API shall treat this structure as opaque. Appropriate APIs, e.g., MP_GetObjectType and MP_GetAssociatedPluginOid, shall be used to extract information from the structure.

6.14 MP_OID_LIST

Format

```
typedef struct _MP_OID_LIST
{
   MP_UINT32         oidCount;
   MP_OID            oids[1];
} MP_OID_LIST;
```

Fields

oidCount
The number of object IDs in the *oids* array.
oids
A variable length array of zero or more object IDs. There are *oidCount* objects IDs in this array.

Remarks

This structure is used by a number of APIs to return lists of objects. Any instance of this structure returned by an API shall be freed by a client using the MP_FreeOidList API.

Although *oids* is declared to be an array of one MP_OID structure it can in fact contain any number of MP_OID structures.

6.15 MP_PORT_TRANSPORT_TYPE

Constants

```
#define MP_PORT_TRANSPORT_TYPE_UNKNOWN     0
#define MP_PORT_TRANSPORT_TYPE_MPNODE      1
#define MP_PORT_TRANSPORT_TYPE_FC          2
#define MP_PORT_TRANSPORT_TYPE_SPI         3
#define MP_PORT_TRANSPORT_TYPE_ISCSI       4
#define MP_PORT_TRANSPORT_TYPE_IFB         5

typedef MP_UINT32 MP_PORT_TRANSPORT_TYPE;
```

Definitions

MP_PORT_TRANSPORT_TYPE_UNKNOWN
The associated port is of an unknown transport type.
MP_PORT_TRANSPORT_TYPE_MPNODE
For initiator ports only, the associated port is known to be a virtual construct of an underlying multipath driver.
MP_PORT_TRANSPORT_TYPE_FC
The associated port represents a Fibre Channel port. The Name for the port should be a port WWN formatted as 16 unseparated hexadecimal digits, with no leading 0x.
MP_PORT_TRANSPORT_TYPE_SPI
The associated port represents a parallel SCSI port.
MP_PORT_TRANSPORT_TYPE_ISCSI
The associated port represents an iSCSI initiator or target port. The port name should be an iSCSI name in “iqn”, “eui”, or “naa” format and include “’,i,0x” followed by an ISID (for initiator ports) or “,t,0x” followed by a TGPID (for target ports).
MP_PORT_TRANSPORT_TYPE_IFB
The associated port represents a mapped Fibre Channel port on an InfiniBand initiator. The name should be formatted as a FC PortWWN.

Remarks

This type serves two purposes. It identifies the type of transport and the format of the PORT_ID property.

6.16 MP_ACCESS_STATE_TYPE

Constants

```
#define MP_ACCESS_STATE_ACTIVE_OPTIMIZED       0h
#define MP_ACCESS_STATE_ACTIVE_NONOPTIMIZED    1h
#define MP_ACCESS_STATE_STANDBY                2h
#define MP_ACCESS_STATE_UNAVAILABLE            3h
#define MP_ACCESS_STATE_TRANSITIONING          Fh
#define MP_ACCESS_STATE_ACTIVE                10h

typedef MP_UINT32 MP_ACCESS_STATE_TYPE;
```

Definitions

MP_ACCESS_STATE_ACTIVE_OPTIMIZED
: "All target ports within a target port group should be capable of immediately accessing the logical unit."

MP_ACCESS_STATE_ACTIVE_NONOPTIMIZED
: "The processing of some … commands may operate with lower performance than they would if the target port were in the active/optimized target port …access state."

MP_ACCESS_STATE_STANDBY
: The logical unit only supports a small set of management commands and no data transfer commands.

MP_ACCESS_STATE_UNAVAILABLE
: "The unavailable target port … access state is intended for situations when the target port accessibility to a logical unit may be severely restricted due to SCSI target device limitations (e.g., hardware errors)."

MP_ACCESS_STATE_TRANSITIONING
: Indicates the target device is in the process of transitioning between access states. This value cannot be specified by a client, but can be exposed to clients as a property of a target port group.

MP_ACCESS_STATE_ACTIVE
: Used when the client is requesting that target port groups be activated (using the MP_SetTPGAccess API) but does not care whether these port groups are given an active optimized or active non-optimized state. This value will not be returned in a property. This value is not defined in ISO/IEC 14776-453 (SPC-3).

NOTE The descriptions above, indicated in quotation marks, are quoted or paraphrased from ISO/IEC 14776-453 (SPC-3).

Remarks

This enumerated type provides the target port (group) states as described in ISO/IEC 14776-453 (SPC-3).

6.17 MP_LOAD_BALANCE_TYPE

Constants

```
#define MP_LOAD_BALANCE_TYPE_UNKNOWN         1<<0,
#define MP_LOAD_BALANCE_TYPE_ROUNDROBIN      1<<1,
#define MP_LOAD_BALANCE_TYPE_LEASTBLOCKS     1<<2,
#define MP_LOAD_BALANCE_TYPE_LEASTIO         1<<3,
#define MP_LOAD_BALANCE_TYPE_DEVICE_PRODUCT 1<<4,
#define MP_LOAD_BALANCE_TYPE_LBA_REGION      1<<5,
#define MP_LOAD_BALANCE_TYPE_FAILOVER_ONLY   1<<6,
#define MP_LOAD_BALANCE_TYPE_PROPRIETARY1   1<<16,
#define MP_LOAD_BALANCE_TYPE_PROPRIETARY2   1<<17
 // additional proprietary types

typedef MP_UINT32 MP_LOAD_BALANCE_TYPE;
```

Definitions

MP_LOAD_BALANCE_TYPE_UNKNOWN
: The load balance object has an unknown type. If the load balance field has this type then, it is most likely an uninitialized object.

MP_LOAD_BALANCE_TYPE_ROUNDROBIN
: Load balancing object type that is associated with the algorithm that performs load balancing in a round robin manner.

MP_LOAD_BALANCE_TYPE_LEASTBLOCKS
: Load balancing object type that is associated with the algorithm that performs load balancing using the least blocks as a criteria to select a path for forwarding the request.

MP_LOAD_BALANCE_TYPE_LEASTIO
: Load balancing object type that is associated with the algorithm that performs load balancing using the least used IO path as a criteria for forwarding the request.

MP_LOAD_BALANCE_TYPE_DEVICE_PRODUCT
: The load balance algorithm is optimized for the device specified in the MP_DEVICE_PRODUCT_PROPERTIES class associated with the logical unit.

MP_LOAD_BALANCE_TYPE_LBA_REGION
: Load balancing object type that is associated with the algorithm that performs load balancing using the sequential stream detection algorithm.

MP_LOAD_BALANCE_TYPE_FAILOVER_ONLY
: Set in MP_DEVICE_PRODUCT_PROPERTIES when the plugin/driver has determined that the device supports SCSI 2 RESERVE/RELEASE. Used by API clients to indicate that SCSI 2 reservations are in use and multipathing is only to be used for failover.

MP_LOAD_BALANCE_TYPE_PROPRIETARYx
: The load balance algorithm is proprietary. This bit mask supports up to sixteen proprietary types.

Remarks

Plugin support for device-type specific load balance types is expressed through instances of MP_DEVICE_PRODUCT_PROPERTIES. If this property is MP_LOAD_BALANCE_-TYPE_DEVICE_PRODUCT then the vendor, product and revision properties of the logical unit shall match those of instances of MP_DEVICE_PRODUCT_PROPERTIES. In contexts where MP_LOAD_BALANCE_TYPE refers to the current or default value, a single flag should be set or the value should be zero if no load balancing algorithm is in use.

See MP_PROPRIETARY_LOAD_BALANCE_PROPERTIES.

6.18 MP_PROPRIETARY_PROPERTY

Format

```
typedef struct _MP_PROPRIETARY_PROPERTY
{
    MP_WCHAR          name[16];
    MP_WCHAR          value[48];
} MP_LIBRARY_PROPERTIES;
```

Fields

name
: A null terminated Unicode string containing the name of the proprietary property.

value
: A null terminated Unicode string containing the value associated with the proprietary property.

Remarks

A name and value for a proprietary property. Arrays of proprietary properties are included in some data structures.

6.19 MP_PROPRIETARY_LOAD_BALANCE_PROPERTIES

Format

```
typedef struct _MP_PROPRIETARY_LOAD_BALANCE_PROPERTIES
{
   MP_LOAD_BALANCE_TYPE      typeIndex;
   MP_WCHAR                  name[256];
   MP_WCHAR                  vendorName[256];
   MP_UINT32                 proprietaryPropertyCount;
   MP_PROPRIETARY_PROPERTY   proprietaryProperties[8];
} MP_PROPRIETARY_LOAD_BALANCE_PROPERTIES;
```

Fields

typeIndex
: The value (65536 or greater) representing a vendor-specific load balance algorithm.

name
: A name for the vendor-specific load-balancing algorithm. This name is only meaningful to a vendor-specific client application.

vendorName
: A name for the vendor associated with the load-balancing algorithm.

proprietaryPropertyCount
: The count of proprietary properties (less that or equal to eight) supported.

proprietaryProperties
: A list of proprietary property name/value pairs.

Remarks

This structure is optional and allows a vendor to add up to 16 vendor-specific load-balance algorithms to the load balance bit maps used in logical unit and plugin properties.

See MP_LOAD_BALANCE_TYPE

6.20 MP_LOGICAL_UNIT_NAME_TYPE

Constants

```
#define MP_LU_NAME_TYPE_UNKNOWN           0
#define MP_LU_NAME_TYPE_VPD83_TYPE1       1
#define MP_LU_NAME_TYPE_VPD83_TYPE2       2
#define MP_LU_NAME_TYPE_VPD83_TYPE3       3
#define MP_LU_NAME_TYPE_DEVICE_SPECIFIC   4

typedef MP_UINT32 MP_LOGICAL_UNIT_NAME_TYPE;
```

Definitions

MP_LOGICAL_UNIT_NAME_TYPE_UNKOWN
: The interpretation of the name for the logical unit is unknown. Use of this value is discouraged and should only be used if the name is derived from some other driver rather than directly from a SCSI Inquiry command.

MP_LU_NAME_TYPE_VPD83_TYPE1
: The name is derived from SCSI **Device Identification VPD page** (i.e., page 83h), Association 0, Type 1.

MP_LU_NAME_TYPE_VPD83_TYPE2
: The name is derived from SCSI **Device Identification VPD page** (i.e., page 83h), Association 0, Type 2.

MP_LU_NAME_TYPE_VPD83_TYPE3
: The name is derived from SCSI **Device Identification VPD page** (i.e., page 83h), Association 0, Type 3.

MP_LU_NAME_TYPE_DEVICE_SPECIFIC
: The name is derived from a device product specific command.

Remarks

ISO/IEC 14776-453 (SPC-3) allows for several different representations of logical unit names. This property is an enumerated type for commonly used formats.

6.21 MP_LIBRARY_PROPERTIES

Format

```
typedef struct _MP_LIBRARY_PROPERTIES
{
   MP_UINT32         supportedMpVersion;
   MP_WCHAR          vendor[256];
   MP_WCHAR          implementationVersion[256];
   MP_CHAR           fileName[256];
   MP_WCHAR          buildTime[256];
} MP_LIBRARY_PROPERTIES;
```

Fields

supportedMpVersion
: The version of the Multipath Management API implemented by the library. The value returned by a library for the API as described in this document is one.

vendor
: A null terminated Unicode string containing the name of the vendor that created the binary version of the library.

implementationVersion
: A null terminated Unicode string containing the implementation version of the library from the vendor specified in *vendor*.

fileName
: A null terminated ASCII string ideally containing the path and file name of the library that is filling in this structure.
If the path cannot be determined then this field will contain only the name (and extension if applicable) of the file of the library. If this cannot be determined then this field shall be an empty string.

buildTime
: The time and date that the library was built.

6.22 MP_AUTOFAILBACK_SUPPORT

Constants

```
#define MP_AUTOFAILBACK_SUPPORT_NONE           0
#define MP_AUTOFAILBACK_SUPPORT_PLUGIN         1
#define MP_AUTOFAILBACK_SUPPORT_MPLU           2
#define MP_AUTOFAILBACK_SUPPORT_PLUGINANDMPLU  3

typedef MP_UINT32 MP_AUTOFAILBACK_SUPPORT;
```

Definitions

MP_AUTOFAILBACK_SUPPORT_NONE
: The implementation does not support auto-failback.

MP_AUTOFAILBACK_SUPPORT_PLUGIN
: The implementation supports auto-failback properties and APIs across the entire plugin.

MP_AUTOFAILBACK_SUPPORT_MPLU
: The implementation supports auto-failback properties and APIs for individual multipath logical units.

MP_AUTOFAILBACK_SUPPORT_PLUGINANDMPLU
: The implementation supports auto-failback properties and APIs for plugins and individual multipath logical units.

Remarks

Auto-failback is the capability of the implementation to discover that a path has reverted to a usable state and to resume using the path. If the implementation supports auto-failback, then it supports the MP_SetFailbackPollingRate API or shall assure

MP_PLUGIN_PROPERTIES failbackPollingRateMax is set to 0 (indicating polling is not performed or the rate is not tunable).

6.23 MP_AUTOPROBING_SUPPORT

Constants

```
#define MP_AUTOPROBING_SUPPORT_NONE            0
#define MP_AUTOPROBING_SUPPORT_PLUGIN          1
#define MP_AUTOPROBING_SUPPORT_MPLU            2
#define MP_AUTOPROBING_SUPPORT_PLUGINANDMPLU   3

typedef MP_UINT32 MP_AUTOPROBING_SUPPORT;
```

Definitions

MP_AUTOPROBING_SUPPORT_NONE
: The implementation does not support auto-probing.

MP_AUTOPROBING_SUPPORT_PLUGIN
: The implementation supports auto-probing properties and APIs across the entire plugin.

MP_AUTOPROBING_SUPPORT_MPLU
: The implementation supports auto-probing properties and APIs for individual multipath logical units.

MP_AUTOPROBING_SUPPORT_PLUGINANDMPLU
: The implementation supports auto-probing properties and APIs for plugins and individual multipath logical units.

Remarks

Auto-probing is the capability of the implementation to discover state changes in paths that are not being used. Paths may not be used because of administrative weight or path override configurations. If the implementation supports auto-probing, then it supports the MP_SetProbingPollingRate API or shall assure MP_PLUGIN_PROPERTIES probingPollingRateMax is set to 0 (indicating polling is not performed or the rate is not tunable).

6.24 MP_PLUGIN_PROPERTIES

Format

```
typedef struct _MP_PLUGIN_PROPERTIES
{
   MP_UINT32          supportedMpVersion;
   MP_WCHAR           vendor[256];
   MP_WCHAR           implementationVersion[256];
   MP_CHAR            fileName[256];
   MP_WCHAR           buildTime[256];
   MP_WCHAR           driverVendor[256];
   MP_CHAR            driverName[256];
   MP_WCHAR           driverVersion[256];
   MP_UINT32          supportedLoadBalanceTypes;
   MP_BOOL            canSetTPGAccess;
   MP_BOOL            canOverridePaths;
   MP_BOOL            exposesPathDeviceFiles;
   MP_CHAR            deviceFileNamespace[256];
   MP_BOOL            onlySupportsSpecifiedProducts;
   MP_UINT32          maximumWeight;
   MP_AUTOFAILBACK_SUPPORT autoFailbackSupport;
   MP_BOOL            pluginAutoFailbackEnabled;
   MP_UINT32          failbackPollingRateMax;
   MP_UINT32          currentFailbackPollingRate;
   MP_AUTOPROBING_SUPPORT autoProbingSupport;
   MP_BOOL            pluginAutoProbingEnabled;
   MP_UINT32          probingPollingRateMax;
```

```
    MP_UINT32          currentProbingPollingRate;
    MP_LOAD_BALANCE_TYPE defaultloadBalanceType
    MP_UINT32                  proprietaryPropertyCount;
    MP_PROPRIETARY_PROPERTY    proprietaryProperties[8];
} MP_PLUGIN_PROPERTIES;
```

Fields

supportedMpVersion
: The version of the Multipath Management API implemented by a plugin. The value returned by a library for the API as described in this document is one.

vendor
: A null terminated Unicode string containing the name of the vendor that created the binary version of the plugin.

implementationVersion
: A null terminated Unicode string containing the implementation version of the plugin from the vendor specified in *vendor*.

fileName
: A null terminated ASCII string ideally containing the path and file name of the plugin that is filling in this structure.
If the path cannot be determined then this field will contain only the name (and extension if applicable) of the file of the plugin. If this cannot be determined then this field will be an empty string.

buildTime
: The time and date that the plugin that is specified by this structure was built.

driverVendor
: A null terminated Unicode string containing the name of the multipath driver vendor associated with this plugin.

driverName
: A null terminated ASCII string containing the name of the multipath driver associated with the plugin.

driverVersion
: A null terminated Unicode string containing the version number of the multipath driver.

supportedLoadBalanceTypes
: A set of flags representing the load balance types (MP_LOAD_BALANCE_TYPES) supported by the plugin/driver as a plugin-wide property.

canSetTPGAccess
: A boolean indicating whether the implementation supports activating target port groups.

canOverridePaths
: A boolean indicating whether the implementations supports overriding paths. Setting this to true indicates MP_SetOverridePath and MP_CancelOverridePath are supported.

exposesPathDeviceFiles
: A boolean indicating whether the implementation exposes (or leaves exposed) device files for the individual paths encapsulated by the multipath device file. This is typically true for MP drivers that sit near the top of the driver stack.

deviceFileNamespace
: A string representing the primary file names the driver uses for multipath logical units, if those filenames do not match the names in A.1. The name is expressing in the following format:
‘*’ represents one or more alphanumeric characters
‘#’ represents a string of consecutive digits (e.g. ‘5’, ‘123’)
‘%’ represents a string of hexadecimal digits (e.g. ‘6101a45’)
‘\’ is an escape character for literal presentation of *, #, or % (e.g. ‘lu\#5’)
any other character is interpreted literally
For example, “/dev/vx/dmp/*”
If the multipath driver creates multipath logical unit device file names in the same manner as OS device files, then this property should be left null.

onlySupportsSpecifiedProducts
: A boolean indicating whether the driver limits multipath capabilities to certain device types. If true, then the driver only provides multipath support to devices exposed

through MP_DEVICE_PRODUCT_PROPERTIES instances. If false, then the driver supports any device that provides standard SCSI logical unit identifiers.

maximumWeight
: Describes the range of administer settable path weights supported by the driver. A driver with no path preference capabilities should set this property to zero. A driver with the ability to enable/disable paths should set this property to 1. Drivers with more weight settings can set the property appropriately.

autoFailbackSupport
: An enumerated type indicating whether the implementation supports auto-failback at the plugin level, the multipath logical unit level, both levels or whether auto-failback is unsupported.

pluginAutoFailbackEnabled
: A boolean indicating that plugin-wide auto-failback is enabled. This property is undefined if autoFailbackSupport is MP_AUTOFAILBACK_SUPPORT_NONE or MP_AUTOFAILBACK_SUPPORT_MPLU.

failbackPollingRateMax
: The maximum plugin-wide polling rate (in seconds) for auto-failback supported by the driver. Undefined if autofailbackSupport is MP_AUTOFAILBACK_SUPPORT_NONE or MP_AUTOFAILBACK_SUPPORT_MPLU. If the plugin/driver supports auto-failback without polling or does not provide a way to set the polling rate, then this shall be set to zero (0).This value is set by the plugin and cannot be modified by users.

currentFailbackPollingRate
: The current plugin-wide auto-failback polling rate (in seconds). Undefined if autofailbackSupport is MP_AUTOFAILBACK_SUPPORT_NONE or MP_AUTOFAILBACK_SUPPORT_MPLU. Cannot be more that failbackPollingRateMax.

autoProbingSupport
: An enumerated type indicating whether the implementation supports auto-probing at the plugin level, the multipath logical unit level, both levels or whether auto-probing is unsupported.

pluginAutoProbingEnabled
: A boolean indicating that plugin-wide auto-probing is enabled. This property is undefined if autoProbingSupport is MP_AUTOPROBING_SUPPORT_NONE or MP_AUTOPROBING_SUPPORT_MPLU.

probingPollingRateMax
: The maximum plugin-wide polling rate (in seconds) for auto-probing supported by the driver. Undefined if autoProbingSupport is MP_AUTOPROBING_SUPPORT_NONE or MP_AUTOPROBING_SUPPORT_MPLU. If the plugin/driver supports auto-probing without polling or does not provide a way to set the probing polling rate, then this shall be set to zero (0). This value is set by the plugin and cannot be modified by users.

currentProbingPollingRate
: The current plugin-wide auto-probing polling rate (in seconds).Undefined if autoProbingSupport is MP_AUTOPROBING_SUPPORT_NONE or MP_AUTOPROBING_SUPPORT_MPLU. Cannot be more that probingPollingRateMax.

defaultLoadBalanceType
: The load balance type that will be used by the driver for devices (without a corresponding MP_DEVICE_PRODUCT_PROPERTIES instance) unless overridden by the administrator. Any logical unit with vendor, product and revision properties matching a MP_DEVICE_PRODUCT_PROPERTIES instance will default to a device-specific load balance type.

proprietaryPropertyCount
: The count of proprietary properties (less that or equal to eight) supported.

proprietaryProperties
: A list of proprietary property name/value pairs.

6.25 MP_DEVICE_PRODUCT_PROPERTIES

Format

```
typedef struct _MP_DEVICE_PRODUCT_PROPERTIES
 {
   MP_CHAR                    vendor[8];
   MP_CHAR                    product[16];
```

```
    MP_CHAR                     revision[4]
    MP_UINT32                   supportedLoadBalanceTypes;
} MP_DEVICE_PRODUCT_PROPERTIES;
```

Fields

vendor
: Eight bytes of ASCII data identifying the vendor of the device product. Corresponds to the VENDOR IDENTIFICATION field in the SCSI INQUIRY response.

product
: Sixteen bytes of ASCII data. Corresponds to the PRODUCT IDENTIFICATION field in the SCSI INQUIRY response. This field can be set with null in all bytes if all devices with the same vendor and revision fields are treated identically by the plugin.

revision
: Four bytes of ASCII data. Corresponds to the PRODUCT REVISION LEVEL field in the SCSI INQUIRY response. This field can be set with null in all bytes if all devices with the same vendor and product fields are treated identically by the plugin.

supportedLoadBalanceTypes
: A set of flags representing the load balance types (MP_LOAD_BALANCE_TYPES) supported by the device product instance.

Remarks

See the remarks under MP_LOAD_BALANCE_TYPE (see 6.17).

6.26 MP_MULTIPATH_LOGICAL_UNIT_PROPERTIES

Format

```
typedef struct _MP_MULTIPATH_LOGICAL_UNIT_PROPERTIES
{
   MP_CHAR          vendor[8];
   MP_CHAR          product[16];
   MP_CHAR          revision[4];
   MP_CHAR          name[256];
   MP_LOGICAL_UNIT_NAME_TYPE nameType;
   MP_CHAR          deviceFileName[256];
   MP_BOOL          asymmetric;
   MP_OID           overridePath;
   MP_LOAD_BALANCE_TYPE currentLoadBalanceType;
   MP_UINT32        logicalUnitGroupID;
   MP_XBOOL         autoFailbackEnabled;
   MP_UINT32        failbackPollingRateMax;
   MP_UINT32        currentFailbackPollingRate;
   MP_XBOOL         autoProbingEnabled;
   MP_UINT32        probingPollingRateMax;
   MP_UINT32        currentProbingPollingRate
   MP_UINT32                proprietaryPropertyCount;
   MP_PROPRIETARY_PROPERTY  proprietaryProperties[8];
} MP_MULTIPATH_LOGICAL_UNIT_PROPERTIES;
```

Fields

vendor
: Eight bytes of ASCII data identifying the vendor of the device product. Corresponds to the VENDOR IDENTIFICATION field in the SCSI INQUIRY response.

product
: Sixteen bytes of ASCII data. Corresponds to the PRODUCT IDENTIFICATION field in the SCSI INQUIRY response. This field can be set with null in byte 0 if all devices with the same vendor field are treated identically by the plugin.

revision
: Four bytes of ASCII data. Corresponds to the PRODUCT REVISION LEVEL from the SCSI standard inquiry response. This field can be set with null in byte 0 if all devices with the same vendor and product fields are treated identically by the plugin.

name

The name of the device derived from SCSI Inquiry data If the name is derived from SCSI **Device Identification VPD page (i.e., page 83h)** and the CODE SET field is 1 (binary), it is translated to hexadecimal-encoded binary.

nameType
The source of the name property.

deviceFileName
The name of the device file representing the consolidated multi-path device. This name shall comply with A.3.

asymmetric
A boolean indicating whether the underlying logical unit has asymmetric access.

overridePath
The ID of a path object only set when an administrator explicitly sets a path.

currentloadBalanceType
The current load balancing preference assigned to this logical unit.

logicalUnitGroupID
The identifier shared by all logical units in a target device that always shared a common access state. If an API request (MP_SetTPGAccess, MP_EnablePath, MP_DisablePath) forces IOs through a Target Port Group with a different access state, then the target device will force all logical units with a common logicalUnitGroupID to the same access state change.
This property shall correspond to the SCSI logical unit group identifier in a SCSI **Device Identification VPD page (i.e., page 83h)** response. If the target device does not support this SCSI identifier and the plugin understands a proprietary technique for determining groups of logical units that share access state, then the plugin/driver shall generate a value that acts equivalently to the SCSI defined logical unit group behavior. If the target does not support the SCSI logical unit group identifier and the plugin knows the target has symmetric access through all ports, then the plugin shall set this property to zero. If the target does not support the SCSI page 83h logical unit group identifier and the plugin does not have proprietary knowledge of logical unit groups, then this shall be set to FFFFFFFFh.

autoFailbackEnabled
MP_TRUE if the administrator has requested that auto-failback be enabled for this multipath logical unit. If the plugin's autoFailbackSupport is MP_AUTOFAIL-BACK_SUPPORT_PLUGINANDMPLU, MP_UNKNOWN is valid and indicates that multipath logical unit has auto-failback enabled if pluginAutoFailbackEnabled is true. Undefined if the plugin's autoFailbackSupport property is MP_AUTOFAIL-BACK_SUPPORT_NONE or MP_AUTOFAILBACK_SUPPORT_PLUGIN.

failbackPollingRateMax
The maximum polling rate (in seconds) supported by the driver. Zero (0) indicates the driver either does not poll for auto-failback or has not provided an interface to set the polling rate for multipath logical units. If this property and the plugin's failbackPollingRateMax are non-zero, this value has precedence for the associate logical unit. Undefined if the plugin's autoFailbackSupport property is MP_AUTOFAIL-BACK_SUPPORT_NONE or MP_AUTOFAILBACK_SUPPORT_PLUGIN.

currentFailbackPollingRate
The current polling rate (in seconds) for auto-failback. This cannot exceed failbackPollingRateMax. If this property and the plugin's currentFailbackPollingRate are non-zero, this value has precedence for the associate logical unit. Undefined if the plugin's autoFailbackSupport property is MP_AUTOFAILBACK_SUPPORT_NONE or MP_AUTOFAILBACK_SUPPORT_PLUGIN.

autofProbingEnabled
MP_TRUE if the administrator has requested that auto-probing be enabled for this multipath logical unit. If the plugin's autoProbingSupport is MP_AUTO-PROBING_SUPPORT_PLUGINANDMPLU, MP_UNKNOWN is valid and indicates that multipath logical unit has auto-Probing enabled if pluginAutoProbingEnabled is true. Undefined if the plugin's autoProbingSupport property is MP_AUTO-PROBING_SUPPORT_NONE or MP_AUTOPROBING_SUPPORT_PLUGIN.

probingPollingRateMax
The maximum polling rate (in seconds) supported by the driver. Zero (0) indicates the driver either does not poll for auto-Probing or has not provided an interface to set the polling rate for multipath logical units. If this property and the plugin's probingPollingRateMax are non-zero, this value has precedence for the associate

logical unit. Undefined if the plugin's autoProbingSupport property is MP_AUTOPROBING_SUPPORT_NONE or MP_AUTOPROBING_SUPPORT_PLUGIN.

currentProbingPollingRate
The current polling rate (in seconds) for auto-probing. This cannot exceed probingPollingRateMax. If this property and the plugin's currentProbingPollingRate are non-zero, this value has precedence for the associate logical unit. Undefined if the plugin's autoProbingSupport property is MP_AUTOPROBING_SUPPORT_NONE or MP_AUTOPROBING_SUPPORT_PLUGIN.

proprietaryPropertyCount
The count of proprietary properties (less that or equal to eight) supported.

proprietaryProperties
A list of proprietary property name/value pairs.

Remarks

MP_MULTIPATH_LOGICAL_UNIT_PROPERTIES represents an aggregation of paths presented as a virtual device to applications (or drivers higher in the stack). Each MP_PATH_LOGICAL_UNIT_PROPERTIES has a set of associated paths (MP_PATH_LOGICAL_UNIT_PROPERTIES).

6.27 MP_PATH_LOGICAL_UNIT_PROPERTIES

Format

```
typedef struct _MP_PATH_LOGICAL_UNIT_PROPERTIES
{
   MP_UINT32                weight;
   MP_PATH_STATE            pathState;
   MP_BOOL                  disabled;
   MP_OID                   initiatorPortOid;
   MP_OID                   targetPortOid;
   MP_OID                   logicalUnitOid;
   MP_UINT64                logicalUnitNumber;
   MP_CHAR                  deviceFileName[256];
   MP_UINT32                busNumber;
   MP_UINT32                portNumber;
} MP_PATH_LOGICAL_UNIT_PROPERTIES;
```

Fields

weight
The administrator-assigned weight of the path. By default (unless specified by the administrator), all paths are assigned the maximum weight supported by the driver (MP_PLUGIN_PROPERTIES.maximumWeight).

pathState
The path state.

disabled
A boolean indicating that the path is disabled explicitly by the MP_DisablePath API or path weight configuration or implicitly due to path failures.

initiatorPortOid
The object ID of the initiator port associated with the path.

targetPortOid
The object ID of the target port associated with the path.

logicalUnitOid
The object ID of the multipath logical unit associated with the path logical unit.

logicalUnitNumber;
The SCSI logical unit number as a SCSI architecture model (SAM) eight-byte value. Note that in typical cases, the logical unit number.

deviceFileName
The name of the OS device file representing this path, if one exists.

busNumber
On Windows, the bus number associated with the initiator port. Undefined for other platforms.

portNumber

On Windows, the port number associated with the initiator port. Undefined for other platforms.

Remarks

As used throughout this standard, the term “path” applies to a combination of a target port, initiator port and logical unit. Unlike other object/structures defined by this standard, a path does not represent a particular object from the real world, but represents an association between real-world objects. Treating the path as a data-structure allows us to assign it an object ID and treat it like other API objects.

6.28 MP_INITIATOR_PORT_PROPERTIES

Format

```
typedef struct _MP_INITIATOR_PORT_PROPERTIES
{
   MP_CHAR           portID[256];
   MP_PORT_TRANSPORT_TYPEportType;
   MP_CHAR           osDeviceFile[256];
   MP_WCHAR          osFriendlyName[256];
} MP_INITIATOR_PORT_PROPERTIES
```

Fields

portID
: The name of the port. This should be a worldwide unique name defined per transport-specific standards; such as a FC port WWN.

portType
: The transport type of the port.

osDeviceFile
: The OS device file name representing the port on the system. See A.2.

osFriendlyName
: An administrator-friendly name for an initiator port. A name that an administrator would likely use to refer to the port, if known.

Remarks

In order to assure interoperability, portID shall be formatted consistently across implementations.

MP_PORT_TRANSPORT_TYPE_MPNODE	A string representing a platform-specific special device file as described in A.2.
MP_PORT_TRANSPORT_TYPE_FC	A PortWWN formatted as 16 unseparated upper case hex digits (e.g. '21000020372D3C73')
MP_PORT_TRANSPORT_TYPE_SPI	A host/platform name for the port. This is not an interoperable solution, but SPI ports typically lack names.
MP_PORT_TRANSPORT_TYPE_ISCSI	The port name is a string and SHALL be an iSCSI name in “iqn”, “eui”, or “naa” format as described in the iSCSI RFCs.
MP_PORT_TRANSPORT_TYPE_IFB	InfiniBand Global Identifier formatted as 32 unseparated upper case hex digits.

6.29 MP_TARGET_PORT_PROPERTIES

Format

```
typedef struct _MP_TARGET_PORT_PROPERTIES
{
   MP_CHAR           portID[256];
   MP_UINT32         relativePortID
} MP_TARGET_PORT_PROPERTIES
```

Fields

portID
: The name of the port. This should be a worldwide unique name defined in transport-specific standards; such as a FC port WWN.

relativePortID
: An integer identifier for the target port. This corresponds to the relative target port identifier field in a SCSI Management Network Addresses VPD page (I.e., VPD page 85h, see ISO/IEC 14776-453 (SPC-3)) response, type 4h identifier. Note that this value is constrained to 16 bits in ISO/IEC 14776-453 (SPC-3) and that 0 is reserved. If the target device does not support this interface, this property shall be synthesized by the plugin – set this to 1 for port A, 2 for port B, etc.

Remarks

See the remarks above for MP_INITIATOR_PORT_PROPERTIES (see 6.28).

6.30 MP_TARGET_PORT_GROUP_PROPERTIES

Format

```
typedef struct _MP_TARGET_PORT_GROUP_PROPERTIES
{
   MP_ACCESS_STATE_TYPE      accessState;
   MP_BOOL                   explicitFailover;
   MP_BOOL                   supportsLuAssignment;
   MP_BOOL                   preferredLuPath;
   MP_UINT32                 tpgID
} MP_TARGET_PORT_GROUP_PROPERTIES;
```

Fields

accessState
: The access state as defined in ISO/IEC 14776-453 (SPC-3).

explicitFailover
: Set to true if the target device supports an explicit command to set target port group access state (such as the SCSI SET TARGET PORT GROUPS command)

supportsLuAssignment
: A boolean indicating whether the device supports assigning logical units to target port groups. This capability is not based on a standard, but some devices provide this to allow an administrator to optimize throughput by selecting which ports should be used to access specific logical units.

preferredLuPath
: A boolean to identify the preferred path to the associated logical units (PREF bit as described in ISO/IEC 14776-453 (SPC-3)) or a vendor-specific interface.

tpgId
: An integer identifier for the target port group. This corresponds to the TARGET PORT GROUP field in the REPORT TARGET PORT GROUPS response and the TARGET PORT GROUP field in a SCSI **Device Identification VPD page (i.e., page 83h)** response, type 5h identifier.

6.31 MP_TPG_STATE_PAIR

Format

```
typedef struct _MP_TPG_STATE_PAIR
{
   MP_OID                    tpgOid;
   MP_ACCESS_STATE_TYPE      desiredState;
} MP_TPG_STATE_PAIR;
```

Fields

tpgOid
: The object ID of a target port group instance.

state
: The desired state of the target port group.

Remarks

This structure is mandatory if the plugin supports the MP_SetTPGAccess method.

7 APIs

7.1 API overview

APIs to return properties of an object

Many of the APIs return properties of objects. These APIs have names like Puget<object-type>Properties, for example, MP_GetTargetPortProperties.

APIs that associate object instances

Some APIs return object IDs of objects related to another object. For example, MP_GetTargetPortOIDList returns a list of IDs of target port objects that comprise a Target Port Group.

APIs that perform multipath tasks

Includes MP_AssignLogicalUnitToTPG, MP_CancelOverridePath, MP_DisableAutoFailback, MP_DisableAutoProbing, MP_DisablePath, MP_EnableAutoFailback, MP_EnableAutoProbing, MP_EnablePath, MP_SetLogicalUnitLoadBalanceType, MP_SetOverridePath, MP_SetPathWeight, MP_SetPluginLoadBalanceType, MP_SetPollingRate and MP_SetTPGAccess.

Convenience methods

These APIs are not related to multipathing, but provide common programming tasks for clients – MP_CompareOids, MP_FreeOidList, MP_GetAssociatedPluginOid, MP_GetObjectType.

APIs related to installation

MP_DeregisterPlugin and MP_RegisterPlugin

APIs related to events

MP_DeregisterForObjectPropertyChanges, MP_DeregisterForVisibilityChanges, MP_RegisterForObjectPropertyChanges, MP_RegisterForVisibilityChanges,

Typical discovery scenario

A typical client task starts by discovering a subset of the classes by making a sequence of API calls. Once this subset is discovered, the client may display the results or issue another API call to make a change requested by the user. The general discovery pattern in this API is to use an association API to get a list of associated object IDs, then use a properties API on each object ID to get the details.

The diagram below helps a developer understand which API calls are needed for discovery. The dashed lines include the function name that a client will use to determine which other objects (the arrow end of the line) are associated to a given object (the line-end without an arrow).

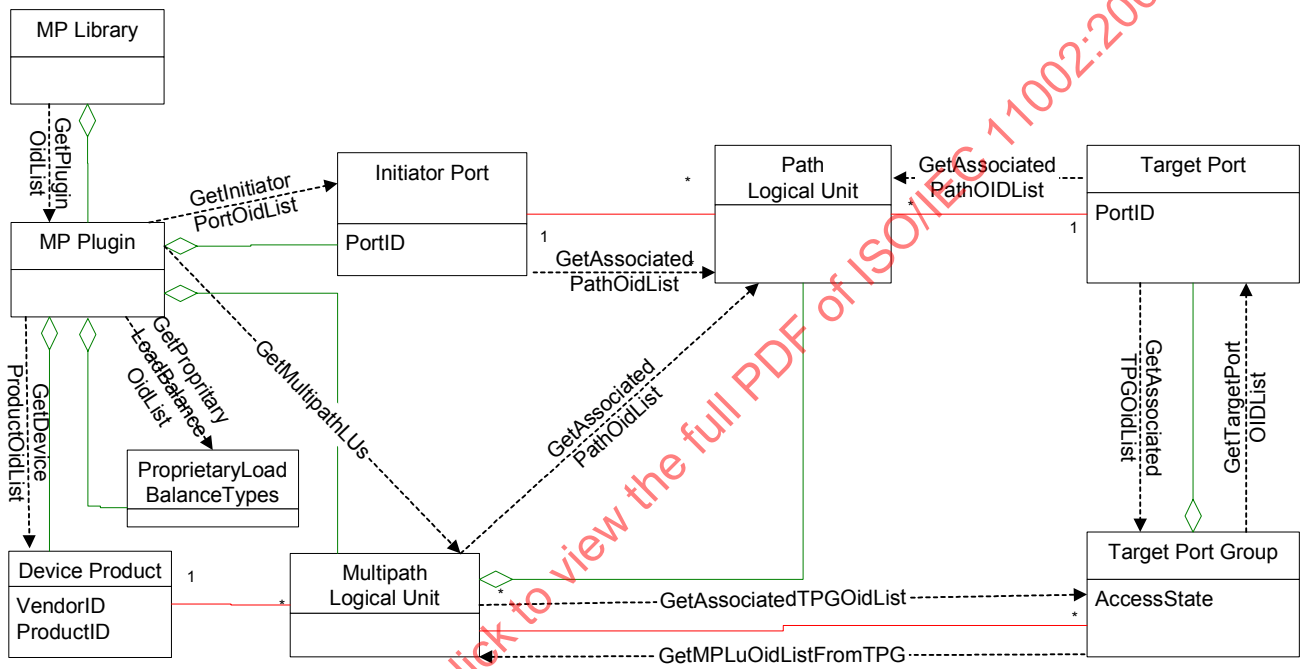

Figure 5 – APIs relative to the objects from Figure 1

Discovery of a model subset typically starts at the library (upper left), finds associated plugins (follow the dashed line) by calling GetPluginOidList, then uses GetPluginProperties to get plugin details. After that, the client has choices which other classes to navigate, depending on the particular task. If the task requires a list of initiator ports, follow the dashed line to initiator ports (call GetInitiatorPortOidList) and get the details using GetInitiatorPortProperties. From initiator ports, GetAssociatedPathOidList returns a list of paths. The same leapfrog approach can be used to determine which API functions are useful in discovering various subsets of the model.

7.2 MP_AssignLogicalUnitToTPG

Synopsis

Assign a multipath logical unit to a target port group.

Prototype

```
MP_STATUS MP_AssignLogicalUnitToTPG(
   /* in */ MP_OID  tpgOid;
   /* in */ MP_OID  luOid;
);
```

Parameters

tpgOid

An MP_TARGET_PORT_GROUP *object ID*. The target port group currently in active access state that the administrator would like the LU assigned to.

luOid

An MP_MULTIPATH_LOGICAL_UNIT object ID.

Typical return values

MP_STATUS_INVALID_OBJECT_TYPE
Returned when *tpgOid* or *luOid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.

MP_STATUS_INVALID_PARAMETER
Returned when *tpgOid* has a type subfield other than MP_OBJECT_TYPE_TARGET_PORT_GROUP or *luOid* has a type subfield other than MP_OBJECT_TYPE_MULTIPATH_LU.

MP_STATUS_OBJECT_NOT_FOUND
Returned when *tpgOid or luOid* owner ID or object sequence number is invalid.

MP_STATUS_UNSUPPORTED
Returned when the API is not supported.

MP_STATUS_SUCCESS
Returned when the operation is successful.

Remarks

Only valid if the target port group supportsLuAssignment is true. This capability is not defined in SCSI standards. In some cases, devices support this capability through non-SCSI interfaces. This method is only used when devices support this capability through vendor-specific SCSI commands.

At any given time, each LU will typically be associated with two target port groups, one in active state and one in standby state. The result of this API will be that the LU associations change to a different pair of target port groups. The caller should specify the object ID of the desired target port group in active access state.

Support

Optional.

See also

MP_GetAssociatedTPGOidList

MP_GetMPLuOidListFromTPG

MP_TARGET_PORT_GROUP_PROPERTIES.supportsLuAssignment

7.3 MP_CancelOverridePath

Synopsis

Cancel a path override and re-enable load balancing.

Prototype

```
MP_STATUS MP_CancelOverridePath(
   /* in */ MP_OID logicalUnitOid;
);
```

Parameters

logicalUnitOid

An MP_MULTIPATH_LOGICAL_UNIT object ID.

Typical return values

MP_STATUS_INVALID_OBJECT_TYPE
Returned when *logicalUnitOid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.

MP_STATUS_INVALID_PARAMETER
Returned when *logicalUnitOid* has a type subfield other than MP_MULTIPATH_LOGICAL_UNIT.
MP_STATUS_OBJECT_NOT_FOUND
Returned when *logicalUnitOid* owner ID or object sequence number is invalid.
MP_STATUS_SUCCESS
Returned when the operation is successful.
MP_STATUS_UNSUPPORTED
Returned when the API is not supported.

Remarks

Only valid if canOverridePaths is true in plugin properties.

The previous load balance configuration and preferences in effect before the path was overridden are restored.

Support

Optional.

See also

MP_SetOverridePath

7.4 MP_CompareOIDs

Synopsis

Compare two object IDs for equality to see whether they refer to the same object.

Prototype

```
MP_STATUS MP_CompareOIDs (
   /* in */ MP_OID  oid1;
   /* in */ MP_OID  oid2;
);
```

Parameters

oid1, oid2
Object IDs for two objects to compare.

Typical return values

MP_STATUS_FAILED
Returned when the object IDs don't compare.
MP_STATUS_SUCCESS
Returned when the two object IDs do refer to the same object.

Remarks

The fields in the two object IDs are compared field-by-field for equality.

Support

Mandatory.

7.5 MP_DeregisterForObjectPropertyChanges

Synopsis

Deregisters a previously registered client function that is to be invoked whenever an object's property changes.

Prototype

```
MP_STATUS MP_DeregisterForObjectPropertyChanges (
   /* in */   MP_OBJECT_PROPERTY_FN   pClientFn,
   /* in */   MP_OBJECT_TYPE          objectType,
```

```
   /* in */   MP_OID                    pluginOid
);
```

Parameters

pClientFn

A pointer to an MP_OBJECT_PROPERTY_FN function defined by the client that was previously registered using the MP_RegisterForObjectPropertyChanges API. On successful return this function will no longer be called to inform the client of object property changes.

objectType

The type of object the client wishes to deregister for property change callbacks. If null, then all object types are deregistered.

pluginOid

If this is a valid plugin object ID, then registration will be removed from that plugin. If this is zero, then registration is removed for all plugins.

Typical return values

MP_STATUS_INVALID_OBJECT_TYPE
Returned when *pluginOid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.

MP_STATUS_INVALID_PARAMETER
Returned when *pluginOid* is not zero and has a type subfield other than MP_OBJECT_TYPE_PLUGIN.

MP_STATUS_OBJECT_NOT_FOUND
Returned when *pluginOid* owner ID or object sequence number is invalid.

MP_STATUS_UNKNOWN_FN
Returned when *pClientFn* is not the same as the previously registered function.

MP_STATUS_SUCCESS
Returned when *pClientFn* is deregistered successfully.

MP_STATUS_FAILED
Returned when *pClientFn* deregistration is not possible at this time.

Support

Mandatory.

Remarks

The function specified by pClientFn takes a single parameter of type MP_OBJECT_PROPERTY_FN.

The function specified by pClientFn will no longer be called whenever an object's property changes.

See also

MP_RegisterForObjectPropertyChanges.

7.6 MP_DeregisterForObjectVisibilityChanges

Synopsis

Deregisters a client function to be called whenever a high level object appears or disappears.

Prototype

```
MP_STATUS MP_DeregisterForObjectCreationChanges (
   /* in */ MP_OBJECT_VISIBILITY_FN pClientFn,
   /* in */ MP_OBJECT_TYPE          objectType,
   /* in */ MP_OID                  pluginOid
);
```

Parameters

pClientFn

A pointer to an MP_OBJECT_VISIBILITY_FN function defined by the client that was previously registered using the MP_RegisterForObjectVisibilityChanges API. On successful return this function will no longer be called to inform the client of object visibility changes.

objectType

The type of object the client wishes to deregister for visibility change callbacks. If null, then all objects types are deregistered.

pluginOid

If this is a valid plugin object ID, then registration will be removed from that plugin. If this is zero, then registration is removed for all plugins.

Typical return values

MP_STATUS_INVALID_OBJECT_TYPE
Returned when *pluginOid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.

MP_STATUS_INVALID_PARAMETER
Returned when *pluginOid* is not zero or has a type subfield other than MP_OBJECT_TYPE_PLUGIN.

MP_STATUS_OBJECT_NOT_FOUND
Returned when *pluginOid* owner ID or object sequence number is invalid.

MP_STATUS_UNKNOWN_FN
Returned when *pClientFn* is not the same as a previously registered function.

MP_STATUS_SUCCESS
Returned when *pClientFn* is deregistered successfully.

MP_STATUS_FAILED
Returned when *pClientFn* deregistration is not possible at this time

Support

Mandatory.

Remarks

The function specified by pClientFn takes a single parameter of type MP_OBJECT_VISIBILITY_FN.

The function specified by pClientFn will no longer be called whenever high level objects appear or disappear.

See also

MP_RegisterForObjectVisibilityChanges.

7.7 MP_DeregisterPlugin

Synopsis

Deregisters a plugin from the common library.

Prototype

```
MP_STATUS MP_DeregisterPlugin (
   /* in */ MP_WCHAR *pPluginId
);
```

Parameters

pPluginId

A pointer to a Plugin ID previously registered using the MP_RegisterPlugin API.

Typical return values

MP_STATUS_INVALID_PARAMETER
Returned when *pPluginId* is null or specifies a memory area that is not executable.

MP_STATUS_UNKNOWN_FN
Returned when *pPluginId* is not the same as a previously registered function.

MP_STATUS_SUCCESS
Returned when *pPluginId* is deregistered successfully.
MP_STATUS_FAILED
Returned when *pPluginId* deregistration is not possible at this time

Support

Mandatory.

Remarks

The plugin will no longer be invoked by the common library. This API does not dynamically remove the plugin from a running library instance. Instead, it prevents an application that is currently not using a plugin from accessing the plugin. This is generally the behavior expected from dynamically loaded modules.

This API will typically be used during plugin deinstallation or upgrade.

Unlike some other APIs, this API is implemented entirely in the common library.

See also

MP_RegisterPlugin

7.8 MP_DisableAutoFailback

Synopsis

Disables auto-failback for the specified plugin or multipath logical unit.

Prototype

```
MP_STATUS MP_DisableAutoFailback(
   /* in */ MP_OID oid
);
```

Parameters

oid

The object ID of the plugin or the multipath logical unit.

Typical Return Values

MP_STATUS_INVALID_OBJECT_TYPE
Returned when *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.
MP_STATUS_INVALID_PARAMETER
Returned when *oid* has a type subfield other than MP_OBJECT_TYPE_PLUGIN or MP_OBJECT_TYPE_MULTIPATH_LU.
MP_STATUS_OBJECT_NOT_FOUND
Returned when *oid* owner ID or object sequence number is invalid.
MP_STATUS_SUCCESS
Returned when the operation is successful
MP_STATUS_UNSUPPORTED
Returned when the API is not supported.

Support

Mandatory if MP_PLUGIN_PROPERTIES.autoFailbackSupported is not MP_AUTOFAILBACK_SUPPORT_NONE.

See also

MP_EnableAutoFailback

7.9 MP_DisableAutoProbing

Synopsis

Disables auto-probing for the specified plugin or multipath logical unit.

Prototype

```
MP_STATUS MP_DisableAutoProbing(
   /* in */ MP_OID oid
);
```

Parameters

oid

The object ID of the plugin or the multipath logical unit.

Typical return values

MP_STATUS_INVALID_OBJECT_TYPE
Returned when *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.

MP_STATUS_INVALID_PARAMETER
Returned when *oid* has a type subfield other than MP_OBJECT_TYPE_PLUGIN or MP_OBJECT_TYPE_MULTIPATH_LU.

MP_STATUS_OBJECT_NOT_FOUND
Returned when *oid* owner ID or object sequence number is invalid.

MP_STATUS_SUCCESS
Returned when the operation is successful.

MP_STATUS_UNSUPPORTED
Returned when the API is not supported.

Support

Mandatory if MP_PLUGIN_PROPERTIES.autoProbingSupported is not MP_AUTOPROBING_SUPPORT_NONE.

See also

MP_EnableAutoProbing.

7.10 MP_DisablePath

Synopsis

Disables a path. This API may cause failover in a logical unit with asymmetric access.

Prototype

```
MP_STATUS MP_DisablePath(
   /* in */ MP_OID oid
);
```

Parameters

oid

The object ID of the path (MP_PATH_LOGICAL_UNIT_PROPERTIES).

Typical return values

MP_STATUS_INVALID_OBJECT_TYPE
Returned when *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.

MP_STATUS_OBJECT_NOT_FOUND
Returned when *oid* owner ID or object sequence number is invalid.

MP_STATUS_INVALID_PARAMETER
Returned when *oid* does not have a type subfield of MP_OBJECT_TYPE_PATH_LU.

MP_STATUS_UNSUPPORTED
Returned when the API is not supported.

MP_STATUS_TRY_AGAIN
Returned when the path cannot be disabled at this time.

MP_STATUS_NOT_PERMITTED
Returned when disabling this path would cause the logical unit to become unavailable. Whether the implementation returns this value or allows the last path to be disabled is implementation specific.
MP_STATUS_SUCCESS
Returned when the operation is successful.

Support

Optional.

Remarks

This API sets MP_PATH_LOGICAL_UNIT_PROPERTIES.disabled to true.

See also

MP_EnablePath.

7.11 MP_EnableAutoFailback

Synopsis

Enables auto-failback.

Prototype

```
MP_STATUS MP_EnableAutoFailback(
   /* in */ MP_OID oid
);
```

Parameters

oid
The object ID of the plugin or multipath logical unit.

Typical return values

MP_STATUS_INVALID_OBJECT_TYPE
Returned when *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.
MP_STATUS_INVALID_PARAMETER
Returned when *oid* has a type subfield other than MP_OBJECT_TYPE_PLUGIN or MP_OBJECT_TYPE_MULTIPATH_LU.
MP_STATUS_OBJECT_NOT_FOUND
Returned when *oid* owner ID or object sequence number is invalid.
MP_STATUS_SUCCESS
Returned when the operation is successful.
MP_STATUS_UNSUPPORTED
Returned when the API is not supported.

Support

Mandatory if MP_PLUGIN_PROPERTIES.autoFailbackSupported is not MP_AUTOFAILBACK_SUPPORT_NONE.

See also

MP_DisableAutoFailback

7.12 MP_EnableAutoProbing

Synopsis

Enables auto-probing.

Prototype

```
MP_STATUS MP_EnableAutoProbing(
   /* in */ MP_OID oid
);
```

Parameters

oid

The object ID of the plugin or multipath logical unit.

Typical return values

MP_STATUS_INVALID_OBJECT_TYPE
Returned when *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.

MP_STATUS_INVALID_PARAMETER
Returned when *oid* has a type subfield other than MP_OBJECT_TYPE_PLUGIN or MP_OBJECT_TYPE_MULTIPATH_LU.

MP_STATUS_OBJECT_NOT_FOUND
Returned when *oid* owner ID or object sequence number is invalid.

MP_STATUS_SUCCESS
Returned when the operation is successful.

MP_STATUS_UNSUPPORTED
Returned when the API is not supported.

Support

Mandatory if MP_PLUGIN_PROPERTIES.autoProbingSupported is not MP_AUTOPROBING_SUPPORT_NONE.

See also

MP_DisableAutoProbing.

7.13 MP_EnablePath

Synopsis

Enables a path. This API may cause failover in a logical unit with asymmetric access.

Prototype

```
MP_STATUS MP_EnablePath(
   /* in */ MP_OID oid
);
```

Parameters

oid

The object ID of the path.

Typical return values

MP_STATUS_INVALID_OBJECT_TYPE
Returned when *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.

MP_STATUS_INVALID_PARAMETER
Returned when *oid* has a type subfield other than MP_OBJECT_TYPE_PATH_LU.

MP_STATUS_OBJECT_NOT_FOUND
Returned when *oid* owner ID or object sequence number is invalid.

MP_STATUS_UNSUPPORTED
Returned when the API is not supported.

MP_STATUS_TRY_AGAIN
Returned when the path cannot be enabled at this time.

MP_STATUS_SUCCESS
Returned when the operation is successful

Support

Optional.

Remarks

This API sets MP_PATH_LOGICAL_UNIT_PROPERTIES.disabled to false.

See also

MP_DisablePath.

7.14 MP_FreeOidList

Synopsis

Frees memory returned by an MP API.

Prototype

```
MP_STATUS MP_FreeOidList(
   /* in */ MP_OID_LIST *pOidList
);
```

Parameters

pOidList

A pointer to an object ID list returned by an MP API. On successful return, the allocated memory is freed.

Typical return values

MP_STATUS_INVALID_PARAMETER
Returned when *pOidList* is null or specifies a memory area to which data cannot be written.

MP_STATUS_SUCCESS
Returned when the operation is successful.

Remarks

Client shall free all MP_OID_LIST structures returned by any API by using this function.

Support

Mandatory.

7.15 MP_GetAssociatedPathOidList

Synopsis

Get a list of object IDs for all the path logical units associated with the specified multipath logical unit, initiator port or target port.

Prototype

```
MP_STATUS MP_GetAssociatedPathOidList (
   /* in */   MP_OID oid,
   /* out */  MP_OID_LIST **ppList
);
```

Parameters

oid

The object ID of the multipath logical unit, initiator port or target port.

ppList

A pointer to a pointer to an MP_OID_LIST structure. On a successful return, this will contain a pointer to an MP_OID_LIST that contains the object IDs of all the paths associated with the specified (multipath) logical unit, initiator port or target port *oid*.

Typical return values

MP_STATUS_INVALID_PARAMETER
Returned when *ppList* is null or specifies a memory area to which data cannot be written or when *oid* has a type subfield other than MP_OBJECT_TYPE_MULTIPATH_LU, MP_OBJECT_TYPE_INITIATOR_PORT or MP_OBJECT_TYPE_TARGET_PORT.

MP_STATUS_INVALID_OBJECT_TYPE
Returned when *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.
MP_STATUS_OBJECT_NOT_FOUND
Returned when *oid* owner ID or object sequence number is invalid.
MP_STATUS_SUCCESS
Returned when the operation is successful.

Remarks

When the caller is finished using the list it shall free the memory used by the list by calling MP_FreeOidList.

Support

Mandatory.

See also

MP_GetPathLogicalUnitProperties.

7.16 MP_GetAssociatedPluginOid

Synopsis

Gets the object ID for the plugin associated with the specified object ID.

Prototype

```
MP_STATUS MP_GetAssociatedPluginOid(
   /* in */   MP_OID oid
   /* out */  MP_OID *pPluginOid
);
```

Parameters

oid

The object ID of an object that has been received from a previous API call.
pPluginOid

A pointer to an MP_OID structure allocated by the caller. On successful return this will contain the object ID of the plugin associated with the object specified by *oid*.

Typical return values

MP_STATUS_INVALID_OBJECT_TYPE
Returned when *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.
MP_STATUS_INVALID_PARAMETER
Returned when *pluginOid* is null or specifies a memory area to which data cannot be written.
MP_STATUS_OBJECT_NOT_FOUND
Returned when *oid* owner ID is invalid.

Remarks

The sequence number subfield of *oid* is not validated since this API is implemented in the common library.

Support

Mandatory.

7.17 MP_GetAssociatedTPGOidList

Synopsis

Get a list of the object IDs containing the target port group associated with the specified multipath logical unit.

Prototype

```
MP_STATUS MP_GetAssociatedTPGOidList(
   /* in */   MP_OID oid,
   /* out */  MP_OID_LIST **ppList
);
```

Parameters

oid

The object ID of the multipath logical unit.

ppList

A pointer to a pointer to an MP_OID_LIST structure. On a successful return, this will contain a pointer to an MP_OID_LIST that contains the object IDs of target port groups associated with the specified logical unit.

Typical return values

MP_STATUS_INVALID_OBJECT_TYPE
Returned when *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.

MP_STATUS_INVALID_PARAMETER
Returned when *ppList* is null or specifies a memory area to which data cannot be written or *oid* has a type subfield other than MP_OBJECT_TYPE_MULTIPATH_LU.

MP_STATUS_OBJECT_NOT_FOUND
Returned when *oid* owner ID or object sequence number is invalid.

MP_STATUS_SUCCESS
Returned when the operation is successful.

MP_STATUS_FAILED
Returned when the target port group list for the specified object ID is not found.

MP_STATUS_INSUFFICIENT_MEMORY
Returned when memory allocation failure occurs.

Remarks

When the caller is finished using the list it shall free the memory used by the list by calling MP_FreeOidList.

Support

Mandatory.

See also

MP_GetTargetPortGroupProperties.

7.18 MP_GetDeviceProductOidList

Synopsis

Gets a list of the object IDs of all the device product properties associated with this plugin.

Prototype

```
MP_STATUS MP_GetDeviceProductOidList(
   /* in */   MP_OID oid,
   /* out */  MP_OID_LIST **ppList
);
```

Parameters

oid

The object ID of the plugin.

ppList

A pointer to a pointer to an MP_OID_LIST structure. On a successful return, this will contain a pointer to an MP_OID_LIST that contains the object IDs of all the device product descriptors associated with the specified plugin.

Typical return values

MP_STATUS_INVALID_OBJECT_TYPE
Returned when *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.
MP_STATUS_INVALID_PARAMETER
Returned when *ppList* is null or specifies a memory area to which data cannot be written *oid* has a type subfield other than MP_OBJECT_TYPE_PLUGIN.
MP_STATUS_OBJECT_NOT_FOUND
Returned when *oid* owner ID or object sequence number is invalid.
MP_STATUS_SUCCESS
Returned when the operation is successful.
MP_STATUS_FAILED
Returned when the plugin for the specified object ID is not found.
MP_STATUS_INSUFFICIENT_MEMORY
Returned when memory allocation failure occurs.
MP_STATUS_UNSUPPORTED
Returned when the API is not supported.

Remarks

When the caller is finished using the list it shall free the memory used by the list by calling MP_FreeOidList.

Support

Required if the driver supports product-specific load balance types.

See also

MP_GetDeviceProductProperties.

7.19 MP_GetDeviceProductProperties

Synopsis

Get the properties of the specified device product.

Prototype

```
MP_STATUS MP_GetDeviceProductProperties(
   /* in */   MP_OID oid,
   /* out */  MP_DEVICE_PRODUCT_PROPERTIES *pProps
);
```

Parameters

oid
The object ID of the device product.
pProps
A pointer to an MP_DEVICE_PRODUCT_PROPERTIES structure allocated by the caller. On successful return this structure will contain the properties of the device product specified by *oid.*

Typical return values

MP_STATUS_INVALID_OBJECT_TYPE
Returned when *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.
MP_STATUS_OBJECT_NOT_FOUND
Returned when *oid* owner ID or object sequence number is invalid.

MP_STATUS_INVALID_PARAMETER
Returned when *pProps* is null or specifies a memory area to which data cannot be written or *oid* has a type subfield other than MP_OBJECT_TYPE_DEVICE_PRODUCT.
MP_STATUS_SUCCESS
Returned when the operation is successful.
MP_STATUS_FAILED
Returned when the plugin for the specified *oid* is not found.
MP_STATUS_UNSUPPORTED
Returned when the implementation does not support the API.

Support

Required if the driver supports product-specific load balance types.

See also

MP_GetDeviceProductOidList.

7.20 MP_GetInitiatorPortOidList

Synopsis

Gets a list of the object IDs of all the initiator ports associated with this plugin.

Prototype

```
MP_STATUS MP_GetInitiatorPortOidList(
   /* in */   MP_OID         oid,
   /* out */  MP_OID_LIST    **ppList
);
```

Parameters

oid
The object ID of the plugin.
ppList
A pointer to a pointer to an MP_OID_LIST structure. On a successful return, this will contain a pointer to an MP_OID_LIST that contains the object IDs of all the initiator ports associated with the specified plugin.

Typical return values

MP_STATUS_INVALID_OBJECT_TYPE
Returned when *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.
MP_STATUS_INVALID_PARAMETER
Returned when *pplist* is null or specifies a memory area to which data cannot be written or when *oid* has a type subfield other than MP_OBJECT_TYPE_PLUGIN.
MP_STATUS_OBJECT_NOT_FOUND
Returned when *oid* owner ID or object sequence number is invalid.
MP_STATUS_SUCCESS
Returned when the operation is successful.
MP_STATUS_INSUFFICIENT_MEMORY
Returned when memory allocation failure occurs.

Remarks

When the caller is finished using the list it shall free the memory used by the list by calling MP_FreeOidList.

Support

Mandatory.

See also

MP_GetInitiatorPortProperties.

7.21 MP_GetInitiatorPortProperties

Synopsis

Gets the properties of the specified initiator port.

Prototype

```
MP_STATUS MP_GetInitiatorPortProperties(
   /* in */   MP_OID   oid,
   /* out */  MP_INITIATOR_PORT_PROPERTIES   *pProps
);
```

Parameters

oid

The object ID of the port.

pProps

A pointer to an MP_INITIATOR_PORT_PROPERTIES structure allocated by the caller. On successful return, this structure will contain the properties of the port specified by *oid*.

Typical return values

MP_STATUS_INVALID_PARAMETER
Returned when *pProps* is null or specifies a memory area to which data cannot be written or when *oid* has a type subfield other than MP_OBJECT_TYPE_INITIATOR_PORT.

MP_STATUS_INVALID_OBJECT_TYPE
Returned when *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.

MP_STATUS_OBJECT_NOT_FOUND
Returned when *oid* owner ID or object sequence number is invalid.

MP_STATUS_SUCCESS
Returned when the operation is successful.

Support

Mandatory.

See also

MP_GetInitiatorPortOidList.

7.22 MP_GetLibraryProperties

Synopsis

Gets the properties of the MP library that is being used.

Prototype

```
MP_STATUS MP_GetLibraryProperties(
   /* out */  MP_LIBRARY_PROPERTIES *pProps
);
```

Parameters

pProps

A pointer to an MP_LIBRARY_PROPERTIES structure allocated by the caller. On successful return this structure will contain the properties of the MP library that is being used.

Typical return values

MP_STATUS_INVALID_PARAMETER
Returned when *pProps* is null or specifies a memory area which cannot be written.

MP_STATUS_SUCCESS
Returned when the operation is successful.

Support

Mandatory.

See also

Example of Getting Library Properties.

7.23 MP_GetMPLuOidListFromTPG

Synopsis

Returns the list of object IDs for multipath logical units associated with the specific target port group.

Prototype

```
MP_STATUS MP_GetMPLuOidListFromTPG(
   /* in */   MP_OID   oid,
   /* out */  MP_OID   **ppList
);
```

Parameters

oid

The object ID of the target port group.

ppList

A pointer to a pointer to an MP_OID_LIST structure. On successful return, this will contain a pointer to an MP_OID_LIST that contains the object IDs of all the (multipath) logical units associated with the specified target port group.

Typical return values

MP_STATUS_INVALID_OBJECT_TYPE
Returned when *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.

MP_STATUS_INVALID_PARAMETER
Returned when *pplist* is null or specifies a memory area to which data cannot be written or when *oid* has a type subfield other than MP_OBJECT_TYPE_TARGET_PORT.

MP_STATUS_OBJECT_NOT_FOUND
Returned when *oid* owner ID or object sequence number is invalid.

MP_STATUS_SUCCESS
Returned when the operation is successful.

MP_STATUS_FAILED
Returned when the multipath logical unit list for the specified target port group object ID is not found

MP_STATUS_INSUFFICIENT_MEMORY
Returned when memory allocation failure occurs.

Remarks

When the caller is finished using the list it shall free the memory used by the list by calling MP_FreeOidList.

Support

Mandatory.

See also

MP_GetMPLogicalUnitProperties.

7.24 MP_GetMPLogicalUnitProperties

Synopsis

Get the properties of the specified logical unit.

Prototype

```
MP_STATUS MP_GetMPLogicalUnitProperties(
   /* in */   MP_OID        oid,
   /* out */  MP_MULTIPATH_LOGICAL_UNIT_PROPERTIES   *pProps
);
```

Parameters

oid

The object ID of the multipath logical unit.

pProps

A pointer to an MP_MULTIPATH_LOGICAL_UNIT_PROPERTIES structure allocated by the caller. On successful return, this structure will contain the properties of the multipath logical unit specified by *oid*.

Typical Return Values

MP_STATUS_INVALID_PARAMETER
Returned when *pProps* is null or specifies a memory area to which data cannot be written or when *oid* has a type subfield other than MP_OBJECT_TYPE_MULTIPATH_LU.

MP_STATUS_INVALID_OBJECT_TYPE
Returned when *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.

MP_STATUS_OBJECT_NOT_FOUND
Returned when *oid* owner ID or object sequence number is invalid.

MP_STATUS_SUCCESS
Returned when the operation is successful

Support

Mandatory.

See also

MP_GetMPLuOidListFromTPG.

MP_GetMultipathLus.

7.25 MP_GetMultipathLus

Synopsis

Returns a list of multipath logical units associated to a plugin.

Prototype

```
MP_STATUS MP_GetMultipathLus(
   /* in */   MP_OID        oid,
   /* out */  MP_OID_LIST   **ppList
);
```

Parameters

oid

The object ID of the plugin or device product object.

ppList

A pointer to a pointer to an MP_OID_LIST structure. On a successful return, this will contain a pointer to an MP_OID_LIST that contains the object IDs of all the (multipath) logical units associated with the specified plugin object ID.

Typical return values

MP_STATUS_INVALID_OBJECT_TYPE
Returned when *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.

MP_STATUS_INVALID_PARAMETER
Returned when *ppList* is null or specifies a memory area that cannot be written or when *oid* has a type subfield other than MP_OBJECT_TYPE_DEVICE_PRODUCT or MP_OBJECT_TYPE_PLUGIN.
MP_STATUS_OBJECT_NOT_FOUND
Returned when *oid* owner ID or object sequence number is invalid.
MP_STATUS_SUCCESS
Returned when the operation is successful.
MP_STATUS_FAILED
Returned when the plugin for the specified object ID is not found.
MP_STATUS_INSUFFICIENT_MEMORY
Returned when memory allocation failure occurs.

Remarks

When the caller is finished using the list it shall free the memory used by the list by calling MP_FreeOidList.

Support

Mandatory.

See also

MP_GetLogicalUnitProperties.

7.26 MP_GetObjectType

Synopsis

Gets the object type of an initialized object ID.

Prototype

```
MP_STATUS MP_GetObjectType(
   /* in */   MP_OID oid,
   /* out */  MP_OBJECT_TYPE *pObjectType
);
```

Parameters

oid
The initialized object ID to get the type of.
pObjectType
A pointer to an MP_OBJECT_TYPE variable allocated by the caller. On successful return it will contain the object type of *oid*.

Typical return values

MP_STATUS_INVALID_OBJECT_TYPE
Returned when *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.
MP_STATUS_INVALID_PARAMETER
Returned when *pObjectType* is null or specifies a memory area to which data cannot be written.
MP_STATUS_SUCCESS
Returned when the operation is successful.

Remarks

This API is provided so that clients can determine the type of object an object ID represents. This can be very useful for a client function that receives notifications.

Support

Mandatory.

See also

MP_RegisterForObjectVisibilityChanges.

7.27 MP_GetPathLogicalUnitProperties

Synopsis

Get the properties of the specified path.

Prototype

```
MP_STATUS MP_GetPathLogicalUnitProperties(
   /* in */   MP_OID oid,
   /* out */  MP_PATH_LOGICAL_UNIT_PROPERTIES *pProps
);
```

Parameters

oid

The object ID of the path logical unit.

pProps

A pointer to an MP_PATH_LOGICAL_UNIT_PROPERTIES structure allocated by the caller. On successful return, this structure will contain the properties of the path specified by *oid*.

Typical return values

MP_STATUS_INVALID_PARAMETER
Returned when *pProps* is null or specifies a memory area to which data cannot be written or when *oid* has a type subfield other than MP_OBJECT_TYPE_PATH_LU.

MP_STATUS_INVALID_OBJECT_TYPE
Returned when *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.

MP_STATUS_OBJECT_NOT_FOUND
Returned when *oid* owner ID or object sequence number is invalid.

MP_STATUS_SUCCESS
Returned when the operation is successful.

Support

Mandatory.

See also

MP_GetAssociatedPathOidList.

7.28 MP_GetPluginOidList

Synopsis

Gets a list of the object IDs of all currently loaded plugins.

Prototype

```
MP_STATUS MP_GetPluginOidList(
   /* out */  MP_OID_LIST **ppList
);
```

Parameters

ppList

A pointer to a pointer to an MP_OID_LIST. On successful return this will contain a pointer to an MP_OID_LIST that contains the object IDs of all of the plugins currently loaded by the library.

Typical return values

MP_STATUS_INVALID_PARAMETER
Returned when *ppList* is null or specifies a memory area to which data cannot be written.

MP_STATUS_SUCCESS
Returned when the operation is successful.
MP_STATUS_FAILED
Returned when the plugin for the specified object ID is not found.
MP_STATUS_INSUFFICIENT_MEMORY
Returned when memory allocation failure occurs.

Remarks

The returned list is guaranteed to not contain any duplicate entries.

When the caller is finished using the list it shall free the memory used by the list by calling MP_FreeOidList.

Support

Mandatory.

See also

MP_FreeOidList.

MP_GetPluginProperties.

Example of Getting Plugin Properties.

7.29 MP_GetPluginProperties

Synopsis

Gets the properties of the specified plugin.

Prototype

```
MP_STATUS MP_GetPluginProperties(
   /* in */   MP_OID oid,
   /* out */  MP_PLUGIN_PROPERTIES *pProps
);
```

Parameters

oid
The object ID of the plugin.
pProps
A pointer to an MP_PLUGIN_PROPERTIES structure allocated by the caller. On successful return, this structure will contain the properties of the plugin specified by *oid*.

Typical return values

MP_STATUS_INVALID_PARAMETER
Returned when *pProps* is null or specifies a memory area to which data cannot be written or when *oid* has a type subfield other than MP_OBJECT_TYPE_PLUGIN.
MP_STATUS_INVALID_OBJECT_TYPE
Returned when *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.
MP_STATUS_OBJECT_NOT_FOUND
Returned when *oid* owner ID or object sequence number is invalid.
MP_STATUS_SUCCESS
Returned when the operation is successful.

Support

Mandatory.

See also

MP_GetProprietaryLoadBalanceProperties.

MP_ GetPluginOidList.

7.30 MP_GetProprietaryLoadBalanceOidList

Synopsis

Gets a list of the object IDs of all the proprietary load balance algorithms associated with this plugin.

Prototype

```
MP_STATUS MP_GetProprietaryLoadBalanceOidList(
   /* in */   MP_OID oid,
   /* out */  MP_OID_LIST **ppList
);
```

Parameters

oid

The object ID of the plugin.

ppList

A pointer to a pointer to an MP_OID_LIST structure. On a successful return, this will contain a pointer to an MP_OID_LIST that contains the object IDs of all the proprietary load balance types associated with the specified plugin.

Typical return values

MP_STATUS_INVALID_OBJECT_TYPE
Returned when *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.

MP_STATUS_INVALID_PARAMETER
Returned when *ppList* is null or specifies a memory area to which data cannot be written or if *oid* has a type subfield other than MP_OBJECT_TYPE_PLUGIN.

MP_STATUS_OBJECT_NOT_FOUND
Returned when *oid* owner ID or object sequence number is invalid.

MP_STATUS_SUCCESS
Returned when the operation is successful.

MP_STATUS_FAILED
Returned when the plugin for the specified object ID is not found.

MP_STATUS_INSUFFICIENT_MEMORY
Returned when memory allocation failure occurs.

MP_STATUS_UNSUPPORTED
Returned when the implementation does not support the API.

Remarks

When the caller is finished using the list it shall free the memory used by the list by calling MP_FreeOidList.

Support

Optional.

See also

MP_GetProprietaryLoadBalanceProperties.

7.31 MP_GetProprietaryLoadBalanceProperties

Synopsis

Get the properties of the specified load balance properties structure.

Prototype

```
MP_STATUS MP_GetProprietaryLoadBalanceProperties (
   /* in */   MP_OID oid,
   /* out */  MP_PROPRIETARY_LOAD_BALANCE_PROPERTIES*pProps
```

```
);
```

Parameters

oid

The object ID of the proprietary load balance structure.

pProps

A pointer to an MP_PROPRIETARY_LOAD_BALANCE_PROPERTIES structure allocated by the caller. On successful return, this structure will contain the properties of the proprietary load balance algorithm specified by *oid*.

Typical return values

MP_STATUS_INVALID_PARAMETER
Returned when *pObjectType* is null or specifies a memory area to which data cannot be written or when *oid* has a type subfield other than MP_OBJECT_TYPE_PROPRIETARY_LOAD_BALANCE.

MP_STATUS_INVALID_OBJECT_TYPE
Returned when *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.

MP_STATUS_OBJECT_NOT_FOUND
Returned when *oid* owner ID or object sequence number is invalid.

MP_STATUS_SUCCESS
Returned when the operation is successful.

Support

Optional.

See also

MP_GetProprietaryLoadBalanceOidList.

7.32 MP_GetTargetPortGroupProperties

Synopsis

Get the properties of the specified target port group.

Prototype

```
MP_STATUS MP_GetTargetPortGroupProperties(
   /* in */  MP_OID oid,
   /* out */ MP_TARGET_PORT_GROUP_PROPERTIES *pProps
);
```

Parameters

oid

The object ID of the target port group.

pProps

A pointer to an MP_TARGET_PORT_GROUP_PROPERTIES structure allocated by the caller. On successful return, this structure will contain the properties of the target port group specified by *oid*.

Typical return values

MP_STATUS_INVALID_PARAMETER
Returned when *pProps* is null or specifies a memory area to which data cannot be written or when *oid* has a type subfield other than MP_OBJECT_TYPE_TARGET_PORT_GROUP.

MP_STATUS_INVALID_OBJECT_TYPE
Returned when *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.

MP_STATUS_OBJECT_NOT_FOUND
Returned when *oid* owner ID or object sequence number is invalid.

MP_STATUS_SUCCESS
Returned when the operation is successful.

Remarks

None.

Support

Mandatory.

See also

MP_GetAssociatedTPGOidList.

7.33 MP_GetTargetPortOidList

Synopsis

Get a list of the object IDs of the target ports in the specified target port group.

Prototype

```
MP_STATUS MP_GetTargetPortOidList(
   /* in */   MP_OID oid,
   /* out */  MP_OID_LIST **ppList
);
```

Parameters

oid

The object ID of the target port group.

ppList

A pointer to a pointer to an MP_OID_LIST structure. On a successful return, this will contain a pointer to an MP_OID_LIST that contains the object IDs of all the target ports associated with the specified target port group *oid*.

Typical return values

MP_STATUS_INVALID_OBJECT_TYPE
Returned when *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.

MP_STATUS_INVALID_PARAMETER
Returned when *ppList* is null or specifies a memory area to which data cannot be written or when *oid* has a type subfield other than MP_OBJECT_TYPE_TARGET_PORT.

MP_STATUS_OBJECT_NOT_FOUND
Returned when *oid* owner ID or object sequence number is invalid.

MP_STATUS_SUCCESS
Returned when the operation is successful.

MP_STATUS_FAILED
Returned when the target port group for the specified object ID is not found.

MP_STATUS_INSUFFICIENT_MEMORY
Returned when memory allocation failure occurs.

Remarks

When the caller is finished using the list it shall free the memory used by the list by calling MP_FreeOidList.

Support

Mandatory.

See also

MP_GetTargetPortProperties.

7.34 MP_GetTargetPortProperties

Synopsis

Gets the properties of the specified target port.

Prototype

```
MP_STATUS MP_GetTargetPortProperties(
   /* in */   MP_OID oid,
   /* out */  MP_TARGET_PORT_PROPERTIES *pProps
);
```

Parameters

oid

The object ID of the port.

pProps

A pointer to an MP_TARGET_PORT_PROPERTIES structure allocated by the caller. On successful return, this structure will contain the properties of the port specified by *oid*.

Typical return values

MP_STATUS_INVALID_PARAMETER
Returned when *pProps* is null or specifies a memory area to which data cannot be written or when *oid* has a type subfield other than MP_OBJECT_TYPE_TARGET_PORT.

MP_STATUS_INVALID_OBJECT_TYPE
Returned when *oid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.

MP_STATUS_OBJECT_NOT_FOUND
Returned when *oid* owner ID or object sequence number is invalid.

MP_STATUS_SUCCESS
Returned when the operation is successful.

Support

Mandatory.

See also

MP_GetTargetPortOidList.

7.35 MP_RegisterForObjectPropertyChanges

Synopsis

Registers a client function to be called whenever the property of an object changes.

Prototype

```
MP_STATUS MP_RegisterForObjectPropertyChanges (
   /* in */ MP_OBJECT_PROPERTY_FN pClientFn,
   /* in */ MP_OBJECT_TYPE        objectType,
   /* in */ void                  *pCallerData,
   /* in */ MP_OID                pluginOid
);
```

Parameters

pClientFn

A pointer to an MP_OBJECT_PROPERTY_FN function defined by the client. On successful return this function will be called to inform the client of objects that have had one or more properties change.

objectType

The type of object the client wishes to register for property change callbacks. If MP_OBJECT_TYPE_UNKNOWN, then all object types are registered.

pCallerData

A pointer that is passed to the callback routine with each event. This may be used by the caller to correlate the event to source of the registration.

pluginOid

If this is a valid plugin object ID, then registration will be limited to that plugin. If this is zero, then the registration is for all plugins.

Typical return values

MP_STATUS_INVALID_OBJECT_TYPE
Returned when *pluginOid* or *objectType* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.
MP_STATUS_OBJECT_NOT_FOUND
Returned when *pluginOid* owner ID or object sequence number is invalid.
MP_STATUS_INVALID_PARAMETER
Returned when *pCallerData* is null or if *pluginOid* has a type subfield other than MP_OBJECT_TYPE_PLUGIN or when objectType is invalid.
MP_STATUS_SUCCESS
Returned when the operation is successful.
MP_STATUS_FN_REPLACED
Returned when an existing client function is replaced with the one specified in pClientFn.

Support

Mandatory.

Remarks

The function specified by pClientFn takes a single parameter of type MP_OBJECT_PROPERTY_FN.

The function specified by pClientFn will be called whenever the property of an object changes. For the purposes of this function a property is defined to be a field in an object's property structure and the object's status. Therefore, the client function will not be called if a statistic of the associated object changes. But, it will be called when the status changes (e.g. from working to failed) or when a name or other field in a property structure changes.

It is not an error to re-register a client function. However, a client function has only one registration. The first call to deregister a client function will deregister it no matter how many calls to register the function have been made.

If multiple properties of an object change simultaneously, a client function may be called only once to be notified that the changes have occurred.

See also

MP_DeregisterForObjectPropertyChanges.

7.36 MP_RegisterForObjectVisibilityChanges

Synopsis

Registers a client function to be called whenever a high level object appears or disappears.

Prototype

```
MP_STATUS MP_RegisterForObjectVisibilityChanges (
   /* in */ MP_OBJECT_VISIBILITY_FN     pClientFn,
   /* in */ MP_OBJECT_TYPE              objectType,
   /* in */ void                        *pCallerData,
   /* in */ MP_OID                      pluginOid
);
```

Parameters

pClientFn

A pointer to an MP_OBJECT_VISIBILITY_FN function defined by the client. On successful return this function will be called to inform the client of objects whose visibility has changed.

objectType

The type of object the client wishes to register for visibility change callbacks. If MP_OBJECT_TYPE_UNKNOWN, then all objects types are registered.

pCallerData

A pointer that is passed to the callback routine with each event. This may be used by the caller to correlate the event to source of the registration.

pluginOid

If this is a valid plugin object ID, then registration will be limited to that plugin. If this is zero, then the registration is for all plugins.

Typical return values

MP_STATUS_INVALID_OBJECT_TYPE
Returned when *pluginOid* or *objectType* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.

MP_STATUS_OBJECT_NOT_FOUND
Returned when *pluginOid* owner ID or object sequence number is invalid.

MP_STATUS_INVALID_PARAMETER
Returned when *pCallerData* is null or *pluginOid* has a type subfield other than MP_OBJECT_TYPE_PLUGIN or when *objectType* is invalid.

MP_STATUS_SUCCESS
Returned when the operation is successful.

MP_STATUS_FN_REPLACED
Returned when an existing client function is replaced with the one specified in pClientFn.

Support

Mandatory.

Remarks

The function specified by pClientFn takes a single parameter of type MP_OBJECT_VISIBILITY_FN.

The function specified by pClientFn will be called whenever objects appear or disappear.

It is not an error to re-register a client function. However, a client function has only one registration. The first call to deregister a client function will deregister it, no matter how many calls to register the function have been made.

See also

MP_DeregisterForObjectVisibilityChanges.

7.37 MP_RegisterPlugin

Synopsis

Registers a plugin with the common library. In a POSIX environment, this may be implemented by adding an entry to a conf file (configuration file). In Windows, it may be accomplished with a registry entry.

Prototype

```
MP_STATUS MP_RegisterPlugin (
   /* in */   MP_WCHAR          *pPluginId,
   /* in */   MP_CHAR           *pFileName
);
```

Parameters

pPluginId

A pointer to the key name shall be the reversed domain name of the vendor followed by "." followed by the vendor specific name for the plugin that uniquely identifies the plugin.

pFileName

The full path to the plugin library.

Typical return values

MP_STATUS_INVALID_PARAMETER
Returned when *pFileName* does not exist.
MP_STATUS_SUCCESS
Returned when the operation is successful.

Support

Mandatory.

Remarks

Unlike some other APIs, this API is implemented entirely in the common library. It shall be called before the common library will invoke a plugin.

This API does not impact, dynamically add or change plugins bound to a running library instance. Instead, it causes an application that is currently not using a plugin to access the specified plugin on future calls to the common library. This is generally the behavior expected from dynamically loaded modules.

This API is typically called by a plugin's installation software to inform the common library the path for the plugin library.

It is not an error to re-register a plugin. However, a plugin has only one registration. The first call to deregister a plugin will deregister it, no matter how many calls to register the plugin have been made.

A vendor may register multiple plugins by using separate plugin IDs and filenames.

See also

MP_DeregisterPlugin.

7.38 MP_SetLogicalUnitLoadBalanceType

Synopsis

Set the multipath logical unit's load balancing policy.

Prototype

```
MP_STATUS MP_SetLogicalUnitLoadBalanceType(
   /* in */ MP_OID logicalUnitoid,
   /* in */ MP_LOAD_BALANCE_TYPE loadBalance
);
```

Parameters

logicalUnitOid
The object ID of the multipath logical unit.
loadBalance
The desired load balance policy for the specified logical unit.

Typical return values

MP_STATUS_INVALID_OBJECT_TYPE
Returned when *logicalUnitOid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.
MP_STATUS_INVALID_PARAMETER
Returned when *loadBalance* is invalid or *logicalUnitOid* has a type subfield other than MP_OBJECT_TYPE_MULTIPATH_LU.
MP_STATUS_OBJECT_NOT_FOUND
Returned when *logicalUnitOid* owner ID or object sequence number is invalid
MP_STATUS_SUCCESS
Returned when the operation is successful.
MP_STATUS_FAILED
Returned when the specified loadBalance type cannot be handled by the plugin. One possible reason is a request to set

MP_LOAD_BALANCE_TYPE_PRODUCT when the specified logical unit has no corresponding MP_DEVICE_PRODUCT_PROPERTIES instance (i.e. the plugin does not have a product-specific load balance algorithm for the LU product).

MP_STATUS_UNSUPPORTED
Returned when the implementation does not support the API.

Remarks

The value shall correspond to one of the supported values in MP_PLUGIN_PROPERTIES.SupportedLogicalUnitLoadBalanceTypes.

Support

Optional.

7.39 MP_SetOverridePath

Synopsis

Manually override the path for a logical unit. The path exclusively used to access the logical unit until cleared. Use MP_CancelOverride to clear the override.

Prototype

```
MP_STATUS MP_SetOverridePath(
   /* in */ MP_OID logicalUnitOid,
   /* in */ MP_OID pathOid
);
```

Parameters

logicalUnitOid
The object ID of the multipath logical unit.

pathOid
The object ID of the path logical unit.

Typical return values

MP_STATUS_INVALID_OBJECT_TYPE
Returned when *logicalUnitOid* or *pathOid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.

MP_STATUS_INVALID_PARAMETER
Returned when *logicalUnitOid* has a type subfield other than MP_OBJECT_TYPE_MULTIPATH_LU or if *pathOid* has an object type other than MP_OBJECT_TYPE_PATH_LU.

MP_STATUS_OBJECT_NOT_FOUND
Returned when *logicalUnitOid* or *pathOid* owner ID or object sequence number is invalid.

MP_STATUS_SUCCESS
Returned when the operation is successful.

MP_STATUS_UNSUPPORTED
Returned when the implementation does not support the API.

MP_STATUS_PATH_NONOPERATIONAL
Returned when the driver cannot communicate through a selected path.

Remarks

This API allows the administrator to disable the driver's load balance algorithm and force all I/O to a specific path. The existing path weight configuration is maintained. If the administrator undoes the override (by calling MP_CancelOverridePath), the driver will start load balancing based on the weights of available paths (and target port group access state for asymmetric devices).

If the multipath logical unit is part of a target with asymmetrical access, executing this command could cause failover.

Support

Optional.

7.40 MP_SetPathWeight

Synopsis

Set the weight to be assigned to a particular path.

Prototype

```
MP_STATUS MP_SetPathWeight(
   /* in */ MP_OID pathOid,
   /* in */ MP_UINT32 weight
);
```

Parameters

logicalUnitOid

The object ID of the path logical unit.

weight

A weight that will be assigned to the path logical unit.

Typical return values

MP_STATUS_INVALID_OBJECT_TYPE
Returned when *pathOid* does not specify any valid object type. This is most likely to happen if an uninitialized object ID is passed to the API.

MP_STATUS_OBJECT_NOT_FOUND
Returned when *pathOid* ownerID or object sequence number is invalid.

MP_STATUS_INVALID_PARAMETER
Returned when *pathOid* has a type subfield other than MP_OBJECT_TYPE_PATH_LU or when the weight parameter is greater than the plugin's maximumWeight property.

MP_STATUS_SUCCESS
Returned when the operation is successful.

MP_STATUS_FAILED
Returned when the operation failed.

MP_STATUS_UNSUPPORTED
Returned when the driver does not support setting path weight.

Support

Optional.

7.41 MP_SetPluginLoadBalanceType

Synopsis

Set the default load balance policy for the plugin.

Prototype

```
MP_STATUS MP_SetPluginLoadBalanceType(
   /* in */ MP_OID oid,
   /* in */ MP_LOAD_BALANCE_TYPE loadBalance
);
```

Parameters

oid

The object ID of the plugin.

loadBalance

The desired default load balance policy for the specified plugin.