Document Number: DSP0216

Date: 2009-4-23

Version: 1.0.0

SM CLP-to-CIM Common Mapping Specification

Document Type: Specification

Document Status: DMTF Standard

Document Language: E

CONTENTS

Tables

Foreword

The *SM CLP-to-CIM Common Mapping Specification* (DSP0216) was prepared by the Server Management Working Group of the DMTF.

DMTF is a not-for-profit association of industry members dedicated to promoting enterprise and systems management and interoperability.

Acknowledgements

The authors wish to acknowledge the following people.

Contributors:

- Aaron Merkin – IBM
- Khachatur Papanyan – Dell
- Perry Vincent – Intel

Participants from the DMTF Server Management Working Group:

- Jon Hass – Dell
- Enoch Suen – Dell
- Jeff Hilland – HP
- Christina Shaw – HP
- John Leung – Intel

Introduction

The *SM CLP-to-CIM Common Mapping Specification* describes the common requirements for mapping SM CLP commands, command options, command option argument values, and command target properties to elements of the Common Information Model (CIM). This specification defines the basis for implementations' conformance to the Command Line Protocol specifications. Mapping requirements specific to a Management Profile are defined in a mapping specification for each Management Profile. The *SM CLP-to-CIM Common Mapping Specification*, combined with per-profile command mapping specifications, provides the detail that developers of CLP Services, CIM Servers, and Class Providers need to build compliant implementations of the Server Management Command Line Protocol (SM CLP).

SM CLP-to-CIM Common Mapping Specification

1 Scope

The *SM CLP-to-CIM Common Mapping Specification* describes the common requirements for mapping commands, command options, command option argument values, and command target properties to elements of the Common Information Model (CIM). This specification defines the basis for implementations' conformance to the Command Line Protocol specifications.

This document assumes that the reader is familiar with the information provided in the following resources:

- *CIM Infrastructure Specification*
- CIM Schema
- *Systems Management Architecture for Server Hardware (SMASH) Command Line Protocol (CLP) Architecture White Paper*
- *Server Management Managed Element Addressing Specification*
- *Server Management Command Line Protocol Specification*

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.

2.1 Approved References

CIM Schema, 2.14

DMTF DSP0200, *CIM Operations over HTTP 1.2.0,*
http://www.dmtf.org/standards/published_documents/DSP200.pdf

DMTF DSP0004, *CIM Infrastructure Specification 2.3.0,*
http://www.dmtf.org/standards/published_documents/DSP0004V2.3_final.pdf

DMTF DSP0214, *Server Management Command Line Protocol Specification 1.0.0*,
http://www.dmtf.org/standards/published_documents/DSP0214.pdf

DMTF DSP0215, *Server Management Managed Element Addressing Specification 1.0.0*,
http://www.dmtf.org/standards/published_documents/DSP0215_1.0.0.pdf

DMTF DSP0201, *Specification for the Representation of CIM in XML 2.2.0*,
http://www.dmtf.org/standards/published_documents/DSP201.pdf

SNIA, *Storage Management Initiative Specification (SMI-S) 1.1.0,*
http://www.snia.org/tech_activities/standards/curr_standards/smi

2.2 Other References

ISO/IEC Directives, Part 2, *Rules for the structure and drafting of International Standards,* http://isotc.iso.org/livelink/livelink.exe?func=ll&objId=4230456&objAction=browse&sort=subtype

Unified Modeling Language (UML) from the Open Management Group (OMG), http://www.uml.org

IETF RFC 2396, *Uniform Resource Identifiers (URI): Generic Syntax*, August 1998, http://www.ietf.org/rfc/rfc2396.txt

IETF RFC 2718, *Guidelines for new URL Schemes*, November 1999, http://www.ietf.org/rfc/rfc2718.txt

IETF RFC 2717, *Registration Procedures for URL Scheme Names*, November 1999, http://www.ietf.org/rfc/rfc2717.txt

World Wide Web Consortium, *Extensible Markup Language (XML) 1.0 (Third Edition)*, February 2004, http://www.w3.org/tr/rec-xml

3 Terms and Definitions

For the purposes of this document, the following terms and definitions apply. For the purposes of this document, the terms and definitions given in DSP0214 and DSP0201 also apply.

3.1
can
used for statements of possibility and capability, whether material, physical, or causal

3.2
cannot
used for statements of possibility and capability, whether material, physical, or causal

3.3
conditional
indicates requirements to be followed strictly in order to conform to the document when the specified conditions are met

3.4
mandatory
indicates requirements to be followed strictly in order to conform to the document and from which no deviation is permitted

3.5
may
indicates a course of action permissible within the limits of the document

3.6
need not
indicates a course of action permissible within the limits of the document

3.7
optional
indicates a course of action permissible within the limits of the document

3.8
referencing profile
indicates a profile that owns the definition of this class and can include a reference to this profile in its "Related Profiles" table

3.9
shall
indicates requirements to be followed strictly in order to conform to the document and from which no deviation is permitted

3.10
shall not
indicates requirements to be followed in order to conform to the document and from which no deviation is permitted

3.11
should
indicates that among several possibilities, one is recommended as particularly suitable, without mentioning or excluding others, or that a certain course of action is preferred but not necessarily required

3.12
should not
indicates that a certain possibility or course of action is deprecated but not prohibited

3.13
unspecified
indicates that this profile does not define any constraints for the referenced CIM element or operation

4 Symbols and Abbreviated Terms

The following symbols and abbreviations are used in this document.

4.1
ABNF
Augmented Backus-Naur Form

4.2
CIM
Common Information Model

4.3
CIM Server
Common Information Model Server

4.4
CLP
Command Line Protocol

4.5
MAP
Manageability Access Point

253 **4.6**
254 **ME**
255 Managed Element

256 **4.7**
257 **MOF**
258 Managed Object File

259 **4.8**
260 **UFcT**
261 User-Friendly class Tag

262 **4.9**
263 **UFiT**
264 User-Friendly instance Tag

265 **4.10**
266 **UFsT**
267 User-Friendly selection Tag

268 5 Overview

269 The *SM CLP-to-CIM Common Mapping Specification* describes the mapping of Server Management
270 Command Line Protocol (SM CLP) commands, command options, command option argument values,
271 and command target properties to elements of the Common Information Model (CIM). This specification
272 defines the basis for an implementation's conformance to the SM CLP. The command mapping
273 specification for each profile defines additional requirements to be met when the profile is supported by
274 the instrumentation.

275 5.1 Evaluating and Applying an SM CLP Command

276 The steps for executing an SM CLP command are as follows:

277 1) Command line is validated against command line grammar.
278 2) Command line is validated against requirements in DSP0214.
279 3) A job is created to track command execution.
280 4) The Resultant Target is resolved to an Object Path.
281 5) The SM CLP commands are mapped to the CIM instrumentation.
282 6) The job is completed.
283 7) The CLP output is produced.

284 The scope of this specification is limited to steps 5 and 6.

285 5.2 SM CLP-to-CIM Command Mapping

286 An SM CLP-to-CIM command mapping is a functional definition of how to interpret an SM CLP command
287 and apply it against underlying CIM instrumentation. This mapping includes the validation of requirements
288 specified in the following sections as well as the requirements specified in the per-profile command
289 mapping specification in which the targets are defined. Each mapping involves the conversion from
290 SM CLP syntax and semantics to CIM elements and operations.

5.3 Addressing Requirements

The SM CLP command target address term identifies the CIM instance to which the command is applied. The SM CLP-to-CIM common mapping is not dependent on particular target address term syntax. Each mapping assumes the Resultant Target of a command has been resolved to an Object Path identifying a single CIM instance.

5.4 Determining Requirements for Supporting Functionality

The following sections describe how requirements are prioritized for profiles and overlapping mapping specifications.

5.4.1 Relative Prioritization of Profile and Mapping Requirements

Management profiles specify requirements for support of CIM elements and operations. The requirement to support an element can be mandatory, optional, or conditional. Command mapping specifications state the requirement for support of each CLP command form for a CIM element. The implementation of these command forms may depend on CIM elements that are not mandatory in the underlying instrumentation. Requirements for support of a CLP command form are conditional; they depend on support for the CIM elements that are required to implement the mapping of the CLP command form. In other words, a requirement to support a particular CLP command form does not create a requirement to support the underlying CIM elements. Rather, if the underlying CIM elements are supported, support for the CLP command form would be required.

5.4.2 Overlapping Mapping Specifications

More than one profile and mapping specification pair may define requirements for supporting a CLP command for a CIM class. For a given CIM instance, the requirements for supporting a CLP command shall be the union of requirements specified by the profiles and mapping specifications supported by an implementation.

6 SM CLP-to-CIM Property Mapping

This section specifies how implementations are to use string tokens to reference properties and property values of instances of the target instance's CIM class.

The string name of CIM class properties defined in the MOF for the corresponding class shall be recognized as property names for a SM CLP command target. Implementations shall allow the use of upper or lower case characters for property names.

Implementations shall use the property name in structured output as a SM CLP output keyword or a CLPXML tag as specified in the DSP0214.

6.1 CIM MOF Property Data Type-Based Value Format

The DSP0004 contains specifications for character string formats that are used when initializing property values in a CIM MOF.

When a property value is specified for a property name that matches the corresponding property in the CIM MOF for the target class, the character string format for the property value shall match the format specified in column 3 in Table 1 when the property has the data type specified in column 1 in Table 1.

6.2 SM CLP Data Type Modifier-Based Value Format

This specification defines a set of string tokens that select a specific value string format for the property that it modifies. The data type modifier type string is appended to a property name, separated only by a single sharp or pound sign character (#).

When a property value is specified for a property name that matches the corresponding property in the CIM MOF for the class and the property is modified using the form "#<modifier>", the implementation shall accept the character string format for the value that corresponds to the property's data type and selected modifier as listed in Table 1.

6.3 CLP-to-CIM Property Value String Format Specifications

Table 1 specifies the character string format to be used to express and interpret values for target properties based on the data type of the property as defined by the CIM class MOF, the corresponding specifications in the DSP0004 Version 2.3, Section 2.2, and SM CLP data type modifiers.

Table 1 – CLP-to-CIM Property Value String Formats

CIM MOF Intrinsic Data Types **SM-CLP Data Type Modifiers**	***CIM Infrastructure Specification 2.3* Interpretation**	***CIM Infrastructure Specification 2.3* Grammar Production Token or SM-CLP Modifier Specification**	**Examples**
uint8	Unsigned 8-bit integer	DSP0004, Appendix A, token = integerValue	01001101b 77 0x4d O75
sint8	Signed 8-bit integer	DSP0004, Appendix A, token = integerValue	-01001101b +77 -0x4d +O75
uint16	Unsigned 16-bit integer	See section 6.3.2.	01001101b 77 0x4d O75
sint16	Signed 16-bit integer	DSP0004, Appendix A, token = integerValue	-01001101b +77 -0x4d +O75
uint32	Unsigned 32-bit integer	DSP0004, Appendix A, token = integerValue	01001101b 77 0x4d O75
sint32	Signed 32-bit integer	DSP0004, Appendix A, token = integerValue	-01001101b +77 -0x4d

CIM MOF Intrinsic Data Types **SM-CLP Data Type Modifiers**	***CIM Infrastructure Specification 2.3* Interpretation**	***CIM Infrastructure Specification 2.3* Grammar Production Token or SM-CLP Modifier Specification**	**Examples**
			+O75
uint64	Unsigned 64-bit integer	DSP0004, Appendix A, token = integerValue	01001101b 77 0x4d O75
sint64	Signed 64-bit integer	DSP0004, Appendix A, token = integerValue	-01001101b +77 -0x4d +O75
string	UCS-2 string	DSP0004, Appendix A, token = stringValue	foo "foo bar"
boolean	Boolean	DSP0004, Appendix A, token = booleanValue	true TRUE False FAlSe
real32	IEEE 4-byte floating-point	DSP0004, Appendix A, token = realValue	-10.234e2 -10.234e-2 1.0234e0
real64	IEEE 8-byte floating-point	DSP0004, Appendix A, token = realValue	-10.234e2 -10.234e-2 1.0234e0
datetime	A string containing a date-time as specified in DSP0004	Section 6.3.1, datetime-absolute	19980525133015.0000000-300
#time	User-friendly string specifying absolute date time	Section 6.3.1, datetime-friendly	12:34:00 1998-05-25.12:34:00 1998-05-25.12:34:00-300
#interval	User-friendly string specifying interval	Section 6.3.1, datetime-interval	12:34:00 0002-11-25.12:34:00
<classname> ref	Strongly-typed CIM classname reference	DSP0214, Section 4.14, token = target-Instance	CIM_System CIM_SettingData
char16	16-bit UCS-2 character	DSP0004, Appendix A, token = charValue	a b ' * <

6.3.1 ABNF Productions for Property Value String Formats

The following are ABNF productions for property value string formats:

```
datetime-absolute = 14DIGIT "." 6DIGIT ("+" / "-") 3DIGIT

datetime-friendly = datetime-interval [ ("+" / "-") 3DIGIT ]

datetime-interval = [4DIGIT "-" 2DIGIT "-" 2DIGIT ":"] 2DIGIT ":" 2DIGIT ":" 2DIGIT
```

6.3.2 Value and ValueMap Qualified Properties

When an integer property does not have the Value and ValueMap qualifiers, the format of the property shall be as specified by the `stringValue` token in DSP0004, Appendix A. When an integer property has the Value and ValueMap qualifiers, the string value contained in the Value qualifier shall be interpreted as identifying the corresponding integer value in the ValueMap qualifier. The ValueMap value may be accepted when it is provided in the format specified by the `stringValue` token in DSP0004, Appendix A.

Examples:

```
RequestedState="Enabled"

RequestedState="2"
```

7 Common Mapping Components

The *SM CLP-to-CIM Common Mapping Specification* defines a set of common patterns or algorithms in SM CLP implementations as reusable functions.

Section 7.1 describes common functions for accomplishing frequently performed sequences when implementing basic SM CLP processing and output generation, such as returning error data.

Common functions for object classes that are common across per-profile command mappings are expressed in pseudo-code syntax in section 7.2.

Common functions used to accomplish standard CIM operations are defined in section 7.3.

Common SM CLP messages are documented in section 7.4.

7.1 Common Mapping Functions

This section defines functions that are used by command mapping behavior pseudo-code when documenting functionality that is reusable for SM CLP functionality, such as producing command output elements.

7.1.1 smAddError

This function is used to associate a CIM_Error instance with the instance of CIM_ConcreteJob that represents operation execution. No associations are defined that can be used to associate the CIM_Error instance. Nor is there an extrinsic method on CIM_ConcreteJob to set the reference. This function is defined to provide a clear marker in the pseudo code for the CIM_Error instance that a mapping intends to define as the top-level CIM_Error instance for the operation.

```
smAddError($job, $error)    {
 <unspecified magic>
}
```

`$job` is the CIM_ConcreteJob instance that represents the overall operation.

`$error` is the CIM_Error instance that is the top level error for the operation.

7.1.2 smCommandCompleted

This function forms the SM CLP Command status elements when a command completes successfully.

```
   Sub smCommandCompleted($job)     {
      <CommandStatus>.status = 0
      <CommandStatus>.status_tag = "COMMAND COMPETED SUCCESSFULLY"
      <CommandStatus>.job_id = $job.InstanceID
   }
```

7.1.3 smCommandExecutionFailed

The function is used to map CIM_Error Instances to SM CLP Command Status Elements.

For more information on how processing errors are handled, see section 3.1.6 ("Command Processing") in the DSP0214.

```
//Map an array of CIM_Error instances to SM CLP Command Status data elements
   sub smCommandExecutionFailed($job, $errors[])     {
      $Error = $errors[0];
      <CommandStatus>.status = 3
      <CommandStatus>.status_tag = COMMAND EXECUTION FAILED
      //DSP1005 constrains the property value to contain the Job ID
      <CommandStatus>.job_id = $job.ElementName

      <CommandStatus>.errtype $Error.ErrorType;
      if (1 != $Error.ErrorType)   {
        <CommandStatus>.errtype_desc = Values[$Error.ErrorType];
      }
      else   {
        <CommandStatus>.errtype_desc = $Error.OtherErrorType;
      }
      <CommandStatus>.cimstat = $Error.CIMStatusCode;
      if (1 != $Error.CIMStatusCode)    {
            <CommandStatus>.cimstat_desc = Values[$Error.CIMStatusCode];
      }
      else   {
        <CommandStatus>.cimstat_desc = $Error.CIMStatusCodeDescription;
      }

      <CommandStatus>.severity = $Error.PerceivedSeverity;
      <CommandStatus>.severity_desc = Values[$Error.PerceivedSeverity];
      <CommandStatus>.probcause = $Error.ProbableCause;
      if (1 != $Error. ProbableCause)   {
          <CommandStatus>.probcause_desc = Values[$Error. ProbableCause];
      }
      else   {
        <CommandStatus>.probcause_desc = $Error. ProbableCauseDescription;
      }
      <CommandStatus>.recmdaction = $Error.RecommendedActions;
       <CommandStatus>.errsource = $Error.ErrorSource;
      <CommandStatus>.errsourceform = $Error.ErrorSourceFormat;
      if(1 != $Error.ErrorSourceFormat) {
        <CommandStatus>.errsourceform_desc = Values[$Error.ErrorSourceFormat];
      }
      else   {
        <CommandStatus>.errsourceform_desc = $Error.OtherErrorSourceFormat;
      }
```

```
        &smMessage(<CommandStatus>, $Error);
        #i = 1;
        for ); #i< $errors.Length; #i++)  {
          smMessage $errors[#i];
        }

    }//end smErrors
```

7.1.4 smConvertToDateTime

This function is used to convert Time, Time#Time, and Time#Interval values to DateTime values to provide consistent errors for poorly formed values of Time, Time#Time, Time#Interval.

A value of "Time" for #expectedFormat indicates that #propertyValueString is expected to be formatted in a manner that is compliant with the production datetime-absolute specified in section 6.3.1.

A value of "Time#Time" for #expectedFormat indicates that #propertyValueString is expected to be formatted in a manner that is compliant with the production datetime-friendly specified in section 6.3.1.

A value of "Time#Interval" indicates that #propertyValueString is expected to be formatted in a manner that is compliant with the production datetime-interval specified in section 6.3.1.

```
sub #requestedtime smConvertToDatetime(#propertyValueString, #expectedFormat)
{
     if ("Time#Time" == #expectedFormat) {
          //validate against section 6.3.1 datetime-friendly,
         if (#valid)   {
              //return datetime value
          }
     }
     else {
          //validate against Section 6.3.1 datetime-interval
          if (#valid)   {
            //convert to datetime interval value and return
          }
     }

     //value is returned if valid, so if we're here the value must be invalid and we
     need to return the error

       $OperationError = smNewInstance("CIM_Error");
       //CIM_ERR_INVALID_PARAMETER
       $OperationError.CIMStatusCode = 4;
       //Other
       $OperationError.ErrorType = 1;
       //Low
       $OperationError.PerceivedSeverity = 2;
       $OperationError.OwningEntity = DMTF:SMCLP;
       $OperationError.MessageID = 0x00000011;
       $OperationError.Message = "A property value is incorrectly formatted.";
       &smAddError($job, $OperationError);
       &smMakeCommandStatus($job);
       smEnd;
}
```

7.1.5 smGetObjectPath

This function is used to convert a Server Management Managed Element address into an object path of a CIM instance. (The mechanism by which an SM ME address is converted to an object path is outside the scope of this specification.)

```
sub $instance-> smGetObjectPath(#address)   {
if (<successfully mapped>)  {
 return $instance->
}
else {
    $OperationError = smNewInstance("CIM_Error");
    //CIM_ERR_NOT_FOUND
    $OperationError.CIMStatusCode = 6;
    //Other
    $OperationError.ErrorType = 1;
    //Low
    $OperationError.PerceivedSeverity = 2;
    &smAddError($job, $OperationError);
    &smMakeCommandStatus($job);
    smEnd;
}
```

`#address` is a string that contains an absolute path defined according to the DSP0215.

7.1.6 smGetSession

This function is used to fetch the instance of CIM_CLPProtocolEndpoint that represents the session that originated the CLP command for which the invoking mapping was applied. The mechanism by which an instance of CIM_CLPProtocolEndpoint is correlated to a particular CLP session is undefined.

```
sub $instance smGetSession()     {
 <unspecified magic>
}
```

An instance of CIM_CLPProtocolEndpoint is returned.

7.1.7 smMakeCommandStatus

This function creates the Command Status portion of a Command Response.

```
sub smMakeCommandStatus($job) {
    //assume the job has stopped ( normally or otherwise )
    //if the OperationalStatus is not okay, then there should be an instance of
    CIM_Error available through
    //GetError()
      if (2  != $job.OperationalStatus)     {
          %InArguments[] = { }
          %OutArguments[] = {newArgument("Job",
       $instanceConcreteJob.getObjectPath())}
          #Error = smOpInvokeMethod($job,
                          "GetError"
                          %InArguments,
                          %OutArguments,
                          #returncode);

          //Method invocation failed, internal processing error
          if ( 0 != #Error.code || 0 != #returncode )    {
              <CommandStatus>.status = 3
```

```
                <CommandStatus>.status_tag = COMMAND EXECUTION FAILED
                <CommandStatus>.job_id = $job.InstanceID
                <CommandStatus>.errtype = 4;
                <CommandStatus>.errtype_desc = "Software Error";
                <CommandStatus>.cimstat = 1;
                <CommandStatus>.cimstat_desc = "CIM_ERR_FAILED";
                <CommandStatus>.severity = 0;
                <CommandStatus>.severity_desc = "Unknown";
                <CommandStatus>.messages[<CommandStatus>.messages.length].message =
                "An internal software error has occurred.";
                <CommandStatus>.messages[<CommandStatus>.messages.length].owningentit
                y = "DMTF:SMCLP";
                <CommandStatus>.messages[<CommandStatus>.messages.length].message_id
                = 0x00000009;
                smEnd;
            }
            else     {
                //make command status
                $joberror = %OutArguments["Error"];
                &smCommandExecutionFailed($job, {$joberror};
                smEnd;
            }//end if have CIM_Error from GetError()
        }
        else   {
            //command completed successfully
          &smCommandCompleted($job);

            smEnd;
        }
    }//end smMakeCommandStatus()
```

7.1.8 smMessage

This function maps CIM_Error message data to SM CLP message elements.

```
//Produce SM CLP Message data element for instance of CIM_Error

    sub smMessage(<CommandStatus>, $Error)     {
        <CommandStatus>.messages[<CommandStatus>.messages.length].message =
        $Error.Message;
        <CommandStatus>.messages[<CommandStatus>.messages.length].owningentity =
        $Error.OwningEntity;
        <CommandStatus>.messages[<CommandStatus>.messages.length].message_id =
        $Error.MessageID;
        #i = 0;
        for );#i < $Error.MessageArguments.Length; #i+++     {
          <CommandStatus>.messages[<CommandStatus>.messages.length].message_arg[#i] =
        $Error.MessageArguments[#i];
        }
    }
```

`<CommandStatus>` is the Command Status instance to which messages will be added.

`$Error` is the instance of CIM_Error that contains the message information.

7.1.9 smNewInstance

This function is used to create a new CIMInstance that is local to the implementation of a mapping. Values for all properties of the instance are initially unassigned.

When using the smOpCreateInstance function to create an instance in the CIMOM, it is first necessary to allocate the template instance in the client space. The smNewInstance function performs this step. The template instance does not exist in the underlying CIMOM. To create an instance in the CIMOM that has

the specified property values, the implementation uses the smOpCreateInstance function, described in section 7.3.5, specifying the template instance created using the smNewInstance function as the parameter.

```
sub $instance smNewInstance(string #className)    {
 <unspecified magic>
}
```

`#className` identifies the CIM class of which an instance will be instantiated.

7.1.10 smSortInstancePaths

This function sorts a CIMObjectPath array based on the property in the order specified and returns a sorted CIMObjectPath array.

```
Sub $sortedInstancePaths[] smSortInstancePaths ( $instancePaths->[], string
#propertyNameSortBy = NULL, boolean #descendingOrder = true ) {
      <algorithm that sorts instances based on the property in the order
specified>
}
```

`#descendingOrder` identifies the order of sorted CIMInstances based on the value of a certain property. The property that is sorted by can be of different data types.

`#descendingOrder=true` represents the specified order for the following data types:

- integer – decreasing numerical value
- boolean – TRUE value first and FALSE value last
- real – decreasing numerical value
- string – decreasing alphanumerical value
- datetime – from earliest date time (the most recent) to the latest date time

7.2 SM CLP-to-CIM Command Mapping Functions for Common Object Classes

The functions described in the following sections implement functionality that is common across multiple object classes for mappings of SM CLP verbs.

7.2.1 smCreateInstance

This function is a wrapper around the intrinsic CreateInstance method that includes mapping to SM CLP Command Status data elements.

```
//PRECONDITIONS
//  1.  $target parameter contains the object to create
//  2.  $job contains the instance of ConcreteJob modeling the CLP operation
sub void smCreateInstance( $target) {
     #Error = smOpCreateInstance( $target);
     if (0 != #Error.code)
     {
          &smProcessOpError (#Error);
          //includes smEnd;
     }
     else {
          //completed successfully
          &smCommandCompleted($job);
          smEnd;
     }
}
```

7.2.2 smDeleteInstance

This function is a wrapper around the intrinsic DeleteInstance method that includes mapping to SM CLP Command Status data elements.

```
//PRECONDITIONS
//  1.  $target-> parameter contains the objectpath of instance to delete
//  2.  $job contains the instance of ConcreteJob modeling the CLP operation
sub void smDeleteInstance( $target) {
     #Error = smOpDeleteInstance( $target);
     if (0 != #Error.code)
     {
          &smProcessOpError (#Error);
          //includes smEnd;
     }
     else {
          //completed successfully
          &smCommandCompleted($job);
          smEnd;
     }
}
```

7.2.3 smDisplayInstance

This function displays the properties of the instance, conforming to the output format specifications described in the DSP0214.

7.2.3.1 Signature without Pseudo-Properties

If the `#propertyNamesToDisplay[]` parameter is NULL, all properties of the instance are displayed. If the `#propertyNamesToDisplay[]` parameter is not NULL, then only those specified properties are displayed.

```
Sub void smDisplayInstance ( $instance, #propertyNamesToDisplay[] )      {
     <unspecified magic>
     //if #propertyNamesToDisplay[] is not NULL, limit output to only those properties
}
```

7.2.3.2 Signature with Pseudo-Properties

If the `#propertyNamesToDisplay[]` parameter is NULL, all properties of the instance are displayed. If the `#propertyNamesToDisplay[]` parameter is not NULL, then only those specified properties are displayed. If #psuedoPropertiesToDisplay is non-NULL, these properties will be appended to the output.

```
Sub void smDisplayInstance ( $instance, #propertyNamesToDisplay[],
#pseudoPropertiesToDisplay[] ) {
     <unspecified magic>
     //if #propertyNamesToDisplay[] is not NULL, limit properties intrinsic to the
instance to only those properties listed
}
```

7.2.4 smProcessOpError

This function implements error processing for an intrinsic operation. The function will construct appropriate command status for the execution.

```
//Preconditions
//  1.  #Error represents the Error object returned from CIM operation function
//  2. global $job contains the CIM_ConcreteJob instance for the operation

sub void smProcessOpError (<Error> #Error ) {
     if (0 != #Error.code)    {
```

```
     //method invocation failed
         if ( (null != #Error.$error) && (null != #Error.$error[0]) )    {
         //if the method invocation contains an embedded error
            //use it for the Error for the overall job
            &smAddError($job, #Error.$error[0]);
            &smMakeCommandStatus($job);
            smEnd;
         }
         else {
             //operation failed, but no detailed error instance, need to make one up
             //make an Error instance and associate with job for Operation
             $OperationError = smNewInstance("CIM_Error");
             //CIM_ERR_FAILED
             $OperationError.CIMStatusCode = #Error.code;
             //Software Error
             $OperationError.ErrorType = 4;
             //Unknown
             $OperationError.PerceivedSeverity = 0;
              $OperationError.OwningEntity = DMTF:SMCLP;
              $OperationError.MessageID = 0x00000009;
              $OperationError.Message = "An internal software error has occurred.";
             &smAddError($job, $OperationError);
             &smMakeCommandStatus($job);
             smEnd;
         }
     }//if CIM op failed
}
```

7.2.5 smRequestStateChange

This function implements a generic invocation of the RequestStateChange( ) method on sub-classes of CIM_EnabledLogicalElement. The function handles the implementation through the production of Command Status.

Note that the Timeout parameter is not used with this version of the method.

```
//PRECONDITIONS
//  1.  $target-> parameter contains the object name of the target instance
//  2.  #requestedState parameter contains the requested state
//  3. global $job contains the CIM_ConcreteJob instance for the operation

sub void smRequestStateChange($target->, string #requestedState)   {
    $instanceConcreteJob = smNewInstance ("CIM_ConcreteJob");

     %InArguments[] = {newArgument("RequestedState", #requestedState),
     newArgument("TimeoutPeriod", NULL) }
     %OutArguments[] = {newArgument("Job", $instanceConcreteJob.getObjectPath())}
    #Error = smOpInvokeMethod ($target->,
                            "RequestStateChange",
                            %InArguments[],
                            %OutArguments[],
                            #returnStatus);
    if (0 != #Error.code)   {
        //method invocation failed
        if ( (null != #Error.$error) && (null != #Error.$error[0]) )       {
          //if the method invocation contains an embedded error
           //use it for the Error for the overall job
           &smAddError($job, #Error.$error[0]);
           &smMakeCommandStatus($job);
           smEnd;
        }
    else if (#Error.code == 17)     {
    //trap for CIM_METHOD_NOT_FOUND
```

```
733     //and make nice Unsupported msg.
734        //unsupported
735        $OperationError = smNewInstance("CIM_Error");
736        //CIM_ERR_NOT_SUPPORTED
737        $OperationError.CIMStatusCode = 7;
738        //Other
739        $OperationError.ErrorType = 1;
740        //Low
741        $OperationError.PerceivedSeverity = 2;
742        $OperationError.OwningEntity = DMTF:SMCLP;
743        $OperationError.MessageID = 0x00000001;
744        $OperationError.Message = "Operation is not supported.";
745        &smAddError($job, $OperationError);
746        &smMakeCommandStatus($job);
747        smEnd;
748     }
749     else {
750      //operation failed, but no detailed error instance, need to make one up
751        //make an Error instance and associate with job for Operation
752           $OperationError = smNewInstance("CIM_Error");
753           //CIM_ERR_FAILED
754           $OperationError.CIMStatusCode = 1;
755           //Software Error
756           $OperationError.ErrorType = 4;
757           //Unknown
758           $OperationError.PerceivedSeverity = 0;
759           $OperationError.OwningEntity = DMTF:SMCLP;
760           $OperationError.MessageID = 0x00000009;
761           $OperationError.Message = "An internal software error has occurred.";
762           &smAddError($job, $OperationError);
763           &smMakeCommandStatus($job);
764           smEnd;
765     }
766        else {
767          //operation failed, but no detailed error instance, need to make one up
768           //make an Error instance and associate with job for Operation
769              $OperationError = smNewInstance("CIM_Error");
770              //CIM_ERR_FAILED
771              $OperationError.CIMStatusCode = 1;
772              //Software Error
773              $OperationError.ErrorType = 4;
774              //Unknown
775              $OperationError.PerceivedSeverity = 0;
776              $OperationError.OwningEntity = DMTF:SMCLP;
777              $OperationError.MessageID = 0x00000009;
778              $OperationError.Message = "An internal software error has occurred.";
779              &smAddError($job, $OperationError);
780              &smMakeCommandStatus($job);
781              smEnd;
782        }
783     }//if CIM op failed
784     else if (0 == #returnStatus)    {
785      //completed successfully
786      &smCommandCompleted($job);
787      smEnd;
788     }
789     else if (0x4096 == #returnStatus)    {
790        //job spawned, need to watch for it to finish
791        //while the jobstate is "Running"
792        while (4 == $instanceConcreteJob.JobState){<busy wait>}
793        if (2 != $job.OperationalStatus) {
794          %InArguments[] = { }
795           %OutArguments[] = {newArgument("Job", $instanceConcreteJob.getObjectPath())}
796           #Error = smOpInvokeMethod($job,
```

```
                        "GetError"
                        %InArguments,
                        %OutArguments,
                        #returncode);
        //Method invocation failed, internal processing error
        if ( (0 != #Error.code) || (0 != #returncode) )   {
        //make an Error instance and associate with job for Operation
           $OperationError = smNewInstance("CIM_Error");
           //CIM_ERR_FAILED
           $OperationError.CIMStatusCode = 1;
           //Software Error
           $OperationError.ErrorType = 4;
           //Unknown
           $OperationError.PerceivedSeverity = 0;
           $OperationError.OwningEntity = DMTF:SMCLP;
           $OperationError.MessageID = 0x00000009;
           $OperationError.Message = "An internal software error has occurred.";
           &smAddError($job, $OperationError);
           &smMakeCommandStatus($job);
           smEnd;
        }
        else    {
           //make command status
           $joberror = %OutArguments["Error"];
           &smCommandExecutionFailed($job, {$joberror};
       }//end if have CIM_Error from GetError()
    }//embedded job not OK
    else {
        //the job ran to completion (we assume)
        &smCommandComplete($job);
       smEnd;
    }
   }//if job spawned
   else if (1 == #returnStatus)    {
      //unsupported
      $OperationError = smNewInstance("CIM_Error");
      //CIM_ERR_NOT_SUPPORTED
      $OperationError.CIMStatusCode = 7;
      //Other
      $OperationError.ErrorType = 1;
      //Low
      $OperationError.PerceivedSeverity = 2;
      $OperationError.OwningEntity = DMTF:SMCLP;
      $OperationError.MessageID = 0x00000001;
      $OperationError.Message = "Operation is not supported.";
      &smAddError($job, $OperationError);
      &smMakeCommandStatus($job);
      smEnd;
   }
   else if (5 == #returnStatus) {
      //unsupported
      $OperationError = smNewInstance("CIM_Error");
      //CIM_ERR_INVALID_PARAMETER
      $OperationError.CIMStatusCode = 4;
      //Other
      $OperationError.ErrorType = 1;
      //Low
      $OperationError.PerceivedSeverity = 2;
      $OperationError.OwningEntity = DMTF:SMCLP;
      $OperationError.MessageID = 0x00000004;
      $OperationError.Message = "One or more parameters specified are invalid.";
      &smAddError($job, $OperationError);
      &smMakeCommandStatus($job);
```

```
860         smEnd;
861      }
862      else if (6 == #returnStatus || 4099 == #returnStatus) {
863       //busy
864         $OperationError = smNewInstance("CIM_Error");
865         //CIM_ERR_FAILED
866         $OperationError.CIMStatusCode = 1;
867         //Other
868         $OperationError.ErrorType = 1;
869         //Low
870         $OperationError.PerceivedSeverity = 2;
871         $OperationError.OwningEntity = DMTF:SMCLP;
872         $OperationError.MessageID = 0x0000000A;
873         $OperationError.Message = "The target is busy and its state cannot be
874         changed.";
875         &smAddError($job, $OperationError);
876         &smMakeCommandStatus($job);
877         smEnd;
878      }
879      else if (4097 == $returnStatus) {
880       //invalid state transition
881         $OperationError = smNewInstance("CIM_Error");
882         //CIM_ERR_FAILED
883         $OperationError.CIMStatusCode = 1;
884         //Other
885         $OperationError.ErrorType = 1;
886         //Low
887         $OperationError.PerceivedSeverity = 2;
888         $OperationError.OwningEntity = DMTF:SMCLP;
889         $OperationError.MessageID = 0x0000000B;
890         $OperationError.Message = "The target cannot transition to the requested state
891         from its current state.";
892         &smAddError($job, $OperationError);
893         &smMakeCommandStatus($job);
894      }
895      else if (2 == #returnStatus || 4 == #returnStatus ||
896               3 == $returnStatus )   {
897       //generic failure
898         $OperationError = smNewInstance("CIM_Error");
899         //CIM_ERR_FAILED
900         $OperationError.CIMStatusCode = 1;
901         //Other
902         $OperationError.ErrorType = 1;
903         //Low
904         $OperationError.PerceivedSeverity = 2;
905         $OperationError.OwningEntity = DMTF:SMCLP;
906         $OperationError.MessageID = 0x00000002;
907         $OperationError.Message = "Failed.   No further information is available.";
908         &smAddError($job, $OperationError);
909         &smMakeCommandStatus($job);
910      }
911      else {
912       //unspecified return code, generic failure
913         $OperationError = smNewInstance("CIM_Error");
914         //CIM_ERR_FAILED
915         $OperationError.CIMStatusCode = 1;
916         //Other
917         $OperationError.ErrorType = 1;
918         //Low
919         $OperationError.PerceivedSeverity = 2;
920         $OperationError.OwningEntity = DMTF:SMCLP;
921         $OperationError.MessageID = 0x00000002;
922         $OperationError.Message = "Failed.   No further information is available.";
923         &smAddError($job, $OperationError);
```

```
        &smMakeCommandStatus($job);
        smEnd;
    }
}//end smRequestStateChange()
```

7.2.6 smReset

This function implements a generic invocation of the Reset( ) method on sub-classes of CIM_LogicalDevice.

```
//  1.  $target-> parameter contains the object name of the target instance
//  2. global $job contains the CIM_ConcreteJob instance for the operation
sub void smReset($target->)    {
    $instanceConcreteJob = smNewInstance (”CIM_ConcreteJob”);
    %InArguments[] = { };
     %OutArguments[] = { };
    #Error = smOpInvokeMethod ($target->,
                           ”Reset”,
                           %InArguments[],
                           %OutArguments[],
                           #returnStatus);
    if (0 != #Error.code)  {
        //method invocation failed
        if ( (null != #Error.$error) && (null != #Error.$error[0]) )       {
          //if the method invocation contains an embedded error
           //use it for the Error for the overall job
           &smAddError($job, #Error.$error[0]);
           &smMakeCommandStatus($job);
           smEnd;
        }
        else {
          //operation failed, but no detailed error instance, need to make one up
           //make an Error instance and associate with job for Operation
              $OperationError = smNewInstance(“CIM_Error”);
              //CIM_ERR_FAILED
              $OperationError.CIMStatusCode = 1;
              //Software Error
              $OperationError.ErrorType = 4;
              //Unknown
              $OperationError.PerceivedSeverity = 0;
              $OperationError.OwningEntity = DMTF:SMCLP;
              $OperationError.MessageID = 0x00000009;
              $OperationError.Message = “An internal software error has occurred.”;
              &smAddError($job, $OperationError);
              &smMakeCommandStatus($job);
              smEnd;
        }
    }
    else if (0 == #returnStatus)    {
     //completed successfully
     &smCommandCompleted($job);
     smEnd;
    }
    else if (1 == #returnStatus)    {
        //unsupported
        $OperationError = smNewInstance(“CIM_Error”);
        //CIM_ERR_NOT_SUPPORTED
        $OperationError.CIMStatusCode = 7;
        //Other
        $OperationError.ErrorType = 1;
        //Low
        $OperationError.PerceivedSeverity = 2;
        $OperationError.OwningEntity = DMTF:SMCLP;
```

```
        $OperationError.MessageID = 0x00000001;
        $OperationError.Message = "Operation is not supported.";
        &smAddError($job, $OperationError);
        &smMakeCommandStatus($job);
        smEnd;
    }
    else {
     //unspecified return code, generic failure
        $OperationError = smNewInstance("CIM_Error");
        //CIM_ERR_FAILED
        $OperationError.CIMStatusCode = 1;
        //Other
        $OperationError.ErrorType = 1;
        //Low
        $OperationError.PerceivedSeverity = 2;
        $OperationError.OwningEntity = DMTF:SMCLP;
        $OperationError.MessageID = 0x00000002;
        $OperationError.Message = "Failed.   No further information is available.";
        &smAddError($job, $OperationError);
        &smMakeCommandStatus($job);
        smEnd;
    }
}//end smReset()
```

7.2.7 smResetRSC

This function implements an invocation of the RequestStateChange( ) method with a RequestedState parameter of "Reset". This function is the generic mapping of the reset verb to an instance of a sub-class of CIM_EnabledLogicalElement. This function uses the RequestStateChange( ) extrinsic method as opposed to the smReset function (see section 7.2.6), which uses the Reset( ) extrinsic method.

```
//PRECONDITIONS
//  1.  $target-> parameter contains the object name of the target instance

sub void smResetRSC($target->)      {
   &smRequestStateChange($target->, "Reset");
   smEnd;
}
```

7.2.8 smSetInstance

This function sets the property values for the specified instance.

```
Sub void smSetInstance ( $instance, string #propertyNames[], string propertyValues[] )
{
     if ( 0 != #propertyName.length ) {
       for #i in #propertyName[] {
         $instance.<#propertyNames[#i]> = #propertyValues[#i]
       }
       #Error = &smOpModifyInstance ( $instance, #propertyNames[] );
       if (0 != #Error.code) {
         &smProcessOpError (#Error);
         //includes smEnd;
       }
       #Error = &smOpGetInstance ( $instance.getObjectPath(), #propertyNames[],
       $outInstance );
     if (0 != #Error.code)
     {
     &smProcessOpError (#Error);
     //includes smEnd;
     }
     &smDisplayInstance ($outInstance, #propertyNames[]);
}
```

7.2.9 smShowAssociationInstances—One Reference

This function enumerates and displays association instances; these instances might be sorted by a specific property value in descending or ascending order. Due to various uses of this function with inconsistent parameter lists, there are multiple versions of the function defined and the signatures do not follow a pattern.

7.2.9.1 Method Signature One—One Reference

This function enumerates and displays all instances of an association class that reference an instance. The association instances are sorted by the value of a property, if one is specified.

Note that because this version of the function does not take a list of properties, the default behavior is to return all mandatory, non-key properties.

```
Sub void smShowAssociationInstances (string #assocClassName, $containerInstancePath->,
string #propertyNameSortBy, boolean #descendingOrder = true ) {

     #Error = &smOpReferences ( $containerInstancePath->, #assocClassName, #className,
     NULL, NULL, NULL, $outAssocInstancePaths->[]);
     if (0 != #Error.code)
     {
          &smProcessOpError (#Error);
          //includes smEnd;
     }
     &smShowInstancesByInstancePaths ( $outAssocInstancePaths->[], #propertyNameSortBy,
     #descendingOrder );
}
```

7.2.9.2 Method Signature Two—One Reference

This function enumerates and displays all instances of an association class that reference an instance. If an array of property names is specified, only those properties will be displayed for each instance of the association.

```
Sub void smShowAssociationInstances (string #assocClassName, $containerInstancePath->,
#propertyNamesToShow[] = null ) {

     if (null == #propertyNamesToShow)   {
          #propertyNamesToShow = {//array of mandatory non-key properties};
     }

     #Error = &smOpReferences ( $containerInstancePath->, #assocClassName, #className,
     NULL, NULL, NULL, $outAssocInstancePaths->[]);
     if (0 != #Error.code)
     {
          &smProcessOpError (#Error);
          //includes smEnd;
     }
     &smShowInstancesByInstancePaths ( $outAssocInstancePaths->[], #propertyNameSortBy,
     #descendingOrder, #propertyNamesToShow[] );
}
```

7.2.9.3 smShowAssociationInstances Method Signature One—Two References

This function enumerates and displays association instances; these instances might be sorted by a specific property value in descending or ascending order. Note that because this version of the function does not take a list of properties, the default behavior is to return all mandatory, non-key properties.

```
sub void smShowAssociationInstances (string #assocClassName, $instancePathA->,
$instancePathB->,string #propertyNameSortBy, boolean #descendingOrder = true ) {

    #Error = &smOpReferences ( $instancePathA->, #assocClassName, #className,  NULL,
    NULL, NULL, $outAssocInstancePathsA->[]);
    if (0 != #Error.code)
    {
        &smProcessOpError (#Error);
        //includes smEnd;
    }
    #Error = &smOpReferences ( $instancePathB->, #assocClassName, #className,  NULL,
    NULL, NULL, $outAssocInstancePathsB->[]);
    if (0 != #Error.code)
    {
        &smProcessOpError (#Error);
        //includes smEnd;
    }
    #outIndex = 0;
    for $instancePathA-> in $outAssocInstancePathsA->[] {
        if ( contains($instancePathA->,$outInstancePathsB->[])    {
            $outAssocInstancePathsBoth->[$outIndex++] = $instancePathA->;
        }
    }
    &smShowInstancesByInstancePaths ( $outAssocInstancePathsBoth->[],
    #propertyNameSortBy, #descendingOrder );
}
```

7.2.9.4 smShowAssociationInstances Method Signature Two—Two References

This function enumerates and displays association instances. If an array of property names is specified, only those properties will be displayed for each instance of the association.

```
sub void smShowAssociationInstances (string #assocClassName, $instancePathA->,
$instancePathB->, #propertyNamesToShow[] = null) {

    if (null == #propertyNamesToShow)   {
        #propertyNamesToShow = {//array of mandatory non-key properties};
    }

    #Error = &smOpReferences ( $instancePathA->, #assocClassName, #className,  NULL,
    NULL, NULL, $outAssocInstancePathsA->[]);
    if (0 != #Error.code)
    {
        &smProcessOpError (#Error);
        //includes smEnd;
    }

    #Error = &smOpReferences ( $instancePathB->, #assocClassName, #className,  NULL,
    NULL, NULL, $outAssocInstancePathsB->[]);
    if (0 != #Error.code)
    {
        &smProcessOpError (#Error);
        //includes smEnd;
    }
```

```
    #outIndex = 0;
    for $instancePathA-> in $outAssocInstancePathsA->[] {
        if ( contains($instancePathA->,$outInstancePathsB->[])    {
            $outAssocInstancePathsBoth->[$outIndex++] = $instancePathA->;
        }
    }
    &smShowInstancesByInstancePaths ( $outAssocInstancePathsBoth->[],
    #propertyNameSortBy, #descendingOrder, #propertyNamesToShow[] );
}
```

7.2.10 smShowInstance

For a specified instance, this function displays all of the property values or a subset of the property values specified using the `#propertyNamesToShow[]` array parameter.

```
Sub void smShowInstance ( $instance, #propertyNamesToShow[] ) {
    &smDisplayInstance ( $instance, #propertyNamesToShow[] );
}
```

7.2.11 smShowInstancePseudoProperties

For a specified instance, this function displays all of the property values or a subset of the property values specified using the `#propertyNamesToShow[]` array parameter. This includes support for annotating referenced properties that have been added to the instance.

```
sub void smShowInstancePseudoProperties( $Object,  #propertynamelist[],
#pseudopropertylist[]) {
    &smDisplayInstance ( $instance, #propertyNamesToShow[],#pseudopropertylist[] );
}
```

7.2.12 smShowInstances

This function enumerates and displays instances; these instances might be sorted by a specific property value in descending or ascending order. Optionally, the `#propertyNamesToShow[]` array parameter can be used to restrict the returned property values to a specific set.

7.2.12.1 Method Signature One

This function displays all instances of the class identified by the className parameter that are associated through instances of the association class identified by the addressAssocClassName parameter with the instance identified by the containerInstancePath parameter. Instances are returned irrespective of the roles the instances play in the association.

```
    Sub void smShowInstances (string #className,
    String #addressAssocClassName,
    string $containerInstancePath->,
    string #propertyNameSortBy,
    boolean #descendingOrder = true,
    #propertyNamesToShow[] = null ) {

    &smShowInstances (#className, #addressAssocClassName, NULL, NULL,
    $containerInstancePath->, #propertyNameSortBy, #descendingOrder,
    #propertyNamesToShow[]);
}
```

7.2.12.2 Method Signature Two

This function displays all instances of the class identified by the className parameter that are associated through instances of the association class identified by the addressAssocClassName parameter with the

instance identified by the containerInstancePath parameter. This version of the function provides the ability to filter instances based on their role in the association.

```
sub void smShowInstances (string #className, string
#addressAssocClassName,
string #role=NULL, string #resultRole=NULL,
$containerInstancePath->, string
#propertyNameSortBy, boolean #descendingOrder = true, #propertyNamesToShow[] ) {
{
     #Error = &smOpAssociators ( $containerInstancePath->, #addressAssocClassName,
     #className,  #role, #resultRole, NULL, $outInstancePaths->[]);
     if (0 != #Error.code)
     {
          &smProcessOpError (#Error);
          //includes smEnd;
     }
     for (#i=0; I < $outInstancePaths->[].length; i++)       {
          &lShowRole($outInstancePaths->[i++];
     }
}
```

7.2.13 smShowInstancesByInstancePaths

This function enumerates and displays instances; these instances might be sorted by a specific property value in descending or ascending order. Optionally, the `#propertyNamesToShow[]` array parameter can be used to restrict the returned property values to a specific set.

```
sub void smShowInstancesByInstancePaths ( $instancePaths->[], string
#propertyNameSortBy, boolean #descendingOrder = true, #propertyNamesToShow[] = null )
{
     if (0 != $instancePaths->[].length() )
     {
          if ( NULL != propertyNameSortBy )
          {
               $sortedInstancePaths->[] = &smSortInstancePaths ( $instancePaths->[],
     #propertyNameSortBy, #descendingOrder );
          for #i in $sortedInstancePaths->[]
               {
                    #Error = &smGetInstance ( sortedInstancePaths->[#i], $instance );
                    if (0 != #Error.code)        {
                         &smProcessOpError (#Error);
                         //includes smEnd;
                    }
                    &smDisplayInstance ( $instance, #propertyNamesToShow[] );
               }
     }
     else     {
          for #i in $instancePaths->[]
          {
               #Error = &smGetInstance ( $instancePaths->[#i], $instance );
               if (0 != #Error.code)
               {
                 &smProcessOpError (#Error);
                 //includes smEnd;
               }
          &smDisplayInstance ( $instance, #propertyNamesToShow[] );
          }
      }
}
```

7.2.14 smShowInstancesPseudoProperties

This function shows instances and any referenced properties that have been attached to the instances.

```
sub void smShowInstancesPseudoProperties( $Objects[], #propertynamelist[],
#pseudopropertylist[]){

     for #i in $Objects[]
     {
          &smDisplayInstance ( $instance,
     #propertyNamesToShow[],#pseudopropertylist[]);
     }
}
```

7.2.15 smStartRSC

This function implements an invocation of the RequestStateChange( ) method with a RequestedState parameter of “Enabled”. This function is the generic mapping of the start verb to an instance of a sub-class of CIM_EnabledLogicalElement.

```
//PRECONDITIONS
//  1.  $target-> parameter contains the object name of the target instance
sub void smStartRSC($target->) {
     &smRequestStateChange($target->, “Enabled”);
     smEnd;
}
```

7.2.16 smStopRSC

This function implements an invocation of the RequestStateChange( ) method with a RequestedState parameter of “Disabled”. This function is the generic mapping of the stop verb to an instance of a sub-class of CIM_EnabledLogicalElement.

```
//PRECONDITIONS
//  1.  $target-> parameter contains the object name of the target instance

sub void smStartRSC($target->) {
     &smRequestStateChange($target->, “Disabled”);
     smEnd;
}
```

7.3 SM CLP-to-CIM Command Mapping Functions for Intrinsic Operations

This section defines wrapper functions for the intrinsic operations defined in DSP0200.

7.3.1 smOpAssociators

The SM CLP-to-CIM command mapping specifications use this function to find CIM instances associated with a target CIM instance. This function corresponds to the Associators intrinsic operation that is defined in section 2.3.2.14 of DSP0200. The function signature is as follows:

```
Sub <ERROR>smOpAssociators(
       [IN] $instancePath->,
       [IN] string #assocClass,
       [IN,OPTIONAL,NULL] string #resultClass = NULL,
       [IN,OPTIONAL,NULL] string #role = NULL,
       [IN,OPTIONAL,NULL] string #resultRole = NULL,
       [IN,OPTIONAL,NULL] string #propertyList[] = NULL,
       [OUT] $associatedInstances>[])
```

<Error> is defined in section A.2.3.

1284 `$instancePath->` is a reference to the CIM Instance that is the root of the search space.

1285 `$sssocClass` identifies an association class. Results are filtered such that instances returned are
1286 associated with `$ObjectName` through instances of this association.

1287 `#resultClass` is an optional filter on the class of instances that will be returned.

1288 `#role` is an optional identifier for the Role that the target instance plays in the association.

1289 `#resultRole` is an optional identifier for the Role that the result instances play in the association.

1290 `propertyList[]` is an array listing the names of the properties that shall be included.
1291 `propertyList[]` shall be either NULL, which indicates that all properties of the instance are included,
1292 or an array of property name strings, which indicates that only the properties specified in the array are
1293 included. `propertyList[]` shall not be an empty array.

1294 `$associatedInstances[]` is an array that, upon successful completion of the function, contains
1295 CIMObjectPath values for CIM instances that meet the filter criteria.

1296 7.3.2 smOpAssociatorNames

1297 The SM CLP-to-CIM command mapping specifications use this function to find CIM instances associated
1298 with a target CIM instance. This function corresponds to the Associators intrinsic operation that is defined
1299 in section 2.3.2.14 of DSP0200. The function signature is as follows:

```
1300 Sub <ERROR>smOpAssociatorNames(
1301       [IN] $instancePath->,
1302       [IN] string #assocClass,
1303       [IN,OPTIONAL,NULL] string #resultClass = NULL,
1304       [IN,OPTIONAL,NULL] string #role = NULL,
1305       [IN,OPTIONAL,NULL] string #resultRole = NULL,
1306       [IN,OPTIONAL,NULL] string #propertyList[] = NULL,
1307       [OUT] $associatedInstancePaths->[])
```

1308 `<Error>` is defined in section A.2.3.

1309 `$instancePath->` is a reference to the CIM Instance that is the root of the search space.

1310 `$sssocClass` identifies an association class. Results are filtered such that instances returned are
1311 associated with `$ObjectName` through instances of this association.

1312 `#resultClass` is an optional filter on the class of instances that will be returned.

1313 `#role` is an optional identifier for the Role that the target instance plays in the association.

1314 `#resultRole` is an optional identifier for the Role that the result instances play in the association.

1315 `propertyList[]` is an array listing the names of the properties which shall be included.
1316 `propertyList[]` shall be either NULL, which indicates that all properties of the instance are included,
1317 or an array of property name strings, which indicates that only the properties specified in the array are
1318 included. `propertyList[]` shall not be an empty array.

1319 `$associatedInstancePaths->[]` is an array that, upon successful completion of the function,
1320 contains CIMObjectPath values for CIM instances that meet the filter criteria.

1321 7.3.3 smOpReferences

1322 This operation is used to enumerate the association objects that refer to a particular target CIM Object
1323 (Class or Instance).

1324 The SM CLP-to-CIM command mapping specifications use this function to find CIM association instances
1325 that refer to a target CIM instance. This function corresponds to the References intrinsic operation that is
1326 defined in section 2.3.2.16 of DSP0200. The function signature is as follows:

```
Sub <ERROR>smOpReferences(
       [IN] $instancePath->,
       [IN,OPTIONAL,NULL] string #resultClass = NULL,
       [IN,OPTIONAL,NULL] string #role = NULL,
       [IN,OPTIONAL,NULL] string #resultRole = NULL,
       [IN,OPTIONAL,NULL] string #propertyList[] = NULL,
       [OUT] $referencedInstances[])
```

`<Error>` is defined in section A.2.3.

`$instancePath->` is a CIMObjectPath that identifies the CIM Instance that is the root of the search space.

`#resultClass` is an optional filter on the class of instances that will be returned.

`#role` is an optional identifier for the Role that the target instance plays in the association.

`#resultRole` is an optional identifier for the Role that the result instances play in the association.

`propertyList[]` is an array listing the names of the properties that shall be included. `propertyList[]` shall be either NULL, which indicates that all properties of the instance are included, or an array of property name strings, which indicates that only the properties specified in the array are included. `propertyList[]` shall not be an empty array.

`$referencedInstances[]` is an array that, upon successful completion of the function, contains CIMObjectPath values for CIM instances that meet the filter criteria.

7.3.4 smOpReferenceNames

This operation is used to enumerate the association objects that refer to a particular target CIM Object (Class or Instance).

The SM CLP-to-CIM command mapping specifications use this function to find CIM association instances that refer to a target CIM instance. This function corresponds to the References intrinsic operation that is defined in section 2.3.2.16 of DSP0200. The function signature is as follows:

```
Sub <ERROR>smOpReferenceNames(
       [IN] $instancePath->,
       [IN,OPTIONAL,NULL] string #resultClass = NULL,
       [IN,OPTIONAL,NULL] string #role = NULL,
       [IN,OPTIONAL,NULL] string #resultRole = NULL,
       [IN,OPTIONAL,NULL] string #propertyList[] = NULL,
       [OUT] $referencedInstancePaths->[])
```

`<Error>` is defined in section A.2.3.

`$instancePath->` is a CIMObjectPath that identifies the CIM Instance that is the root of the search space.

`#resultClass` is an optional filter on the class of instances that will be returned.

`#role` is an optional identifier for the Role that the target instance plays in the association.

`#resultRole` is an optional identifier for the Role that the result instances play in the association.

`propertyList[]` is an array listing the names of the properties that shall be included. `propertyList[]` shall be either NULL, which indicates that all properties of the instance are included, or an array of property name strings, which indicates that only the properties specified in the array are included. `propertyList[]` shall not be an empty array.

`$referencedInstancePaths->[]` is an array that, upon successful completion of the function, contains CIMObjectPath values for CIM instances that meet the filter criteria.

7.3.5 smOpCreateInstance

The SM CLP-to-CIM command mapping specifications use this function to create a CIM instance. This function corresponds to the CreateInstance intrinsic operation that is defined in section 2.3.2.6 of DSP0200. It is assumed that the caller has used the smNewInstance function (see section 7.1.9) to make a local copy of the instance and is invoking this function to perform the update in the CIMOM. Prior to invoking the function, key properties are required to have a value assigned only if the assignment of a specific value is necessary for the clarity of the mapping that is creating the instance.

```
Sub <ERROR>smOpCreateInstance(
    [IN] $instance)
```

`<Error>` is defined in section A.2.3.

`$instance` is a reference to the CIM instance to be created.

7.3.6 smOpDeleteInstance

The SM CLP-to-CIM command mapping specifications use this function to delete a CIM instance. The function also deletes all the association classes that reference the CIM instance. This function corresponds to the DeleteInstance intrinsic operation that is defined in section 2.3.2.4 of DSP0200.

```
Sub <ERROR>smOpDeleteInstance(
    [IN] $instancePath->)
```

`<Error>` is defined in section A.2.3.

`$instancePath->` is a CIMObjectPath that identifies the CIM instance to be deleted.

7.3.7 smOpEnumerateInstances

The SM CLP-to-CIM command mapping specifications get CIM instances by using the smOpEnumerateInstances( ) common function. This function corresponds to the EnumerateInstances intrinsic operation that is defined in section 2.3.2.11 of DSP0200. The function signature is as follows:

```
Sub  <ERROR>smOpEnumerateInstances ( [IN] string #className,
                                     [OUT] string propertyList[] = NULL,
                                     [OUT] $instances[])
```

`<Error>` is defined in section A.2.3.

`ClassName` identifies the target class name that is the basis for the enumeration.

`propertyList[]` is an array listing the names of the properties that shall be included. `propertyList[]` shall be either NULL, which indicates that all properties of the instance are included, or an array of property name strings, which indicates that only the properties specified in the array are included. `propertyList[]` shall not be an empty array.

`$instances` is a reference to an array of returned CIMInstance values.

7.3.8 smOpEnumerateInstanceNames

The SM CLP-to-CIM command mapping specifications get CIM instances by using the smOpEnumerateInstances( ) common function. This function corresponds to the EnumerateInstances intrinsic operation that is defined in section 2.3.2.11 of DSP0200. The function signature is as follows:

```
Sub  <ERROR>smOpEnumerateInstanceNames ( [IN] string #className,
                                         [OUT] $instancePaths->[])
```

`<Error>` is defined in section A.2.3.

`#className` is a string that identifies the target class name that is the basis for the enumeration.

1412 `$instancePaths->[]` is a reference to an array of CIMObjectPath values that identifies the
1413 enumerated instances.

1414 7.3.9 smOpGetInstance

1415 The SM CLP-to-CIM command mapping specifications get CIM instances by using the
1416 smOpGetInstance( )common function. This function corresponds to the GetInstance intrinsic operation
1417 that is defined in section 2.3.2.2 of DSP0200. The function signature is as follows:

```
1418 Sub  <ERROR>smOpGetInstance ( [IN] $instancePath->,
1419                                      [IN] string #propertyList[],
1420                                [OUT] $instance )
```

1421 `<Error>` is defined in section A.2.3.

1422 `$instancePath->` is the CIMObjectPath to the instance to retrieve.

1423 `propertyList[]` is an array listing the names of the properties which shall be included.
1424 `propertyList[]` shall be either NULL, which indicates that all properties of the instance are included,
1425 or an array of property name strings, which indicates that only the properties specified in the array are
1426 included. `propertyList[]` shall not be an empty array.

1427 `$instance` is a reference to the returned CIMInstance.

1428 7.3.10 smOpInvokeMethod

1429 This function is used to invoke an extrinsic method. This function corresponds to an extrinsic method call
1430 using the METHODCALL and METHODRESPONSE XML elements as specified in section 2.3.1 of
1431 DSP0200. The function signature is as follows:

```
1432 Sub <ERROR>smOpInvokeMethod ( [IN] $instancePath->,
1433                             [IN] string methodName,
1434                             [IN] %inArguments[],
1435                             [OUT] %outArguments[],
1436                             [OUT] integer #returnValue)
```

1437 `<Error>` is defined in section A.2.3.

1438 `$instancePath->` identifies the CIM instance on which the method will be invoked.

1439 `#methodName` is the name of the method to invoke.

1440 `%inArguments` is the array of input arguments to the method. These are the parameters where the IN
1441 qualifier has a value of TRUE. The array is indexed by parameter name.

1442 `%outArguments` is the array of output arguments from the method. These are the parameters where the
1443 OUT qualifier has a value of TRUE. The array is indexed by parameter name.

1444 `#returnValue` is the return code from the extrinsic method.

1445 7.3.11 smOpModifyInstance

1446 The SM CLP-to-CIM command mapping specifications modify CIM instances by using the
1447 smOpModifyInstance( ) common function. This function corresponds to the ModifyInstance intrinsic
1448 operation that is defined in section 2.3.2.8 of DSP0200. The function signature is as follows:

```
1449 Sub <ERROR>smOpModifyInstance($modifiedInstance,
1450                               string #propertyList[])
```

1451 `<ERROR>` is defined in section A.2.3.

1452 `$modifiedInstance` is a modified CIMInstance.

1453 `#propertyList[]` is an array listing the names of the modified properties that should be updated.
1454 `propertyList[]` shall not be an empty array or NULL.

1455 7.4 SM CLP-to-CIM Common Messages

1456 Table 2 lists the standard messages defined for the *SM CL-to-CIM Common Mapping Specification*. The
1457 Owning Entity data element must have a value of `DMTF:SMCLP` for all messages defined in this
1458 specification.

1459 **Table 2 – Common SM CLP Messages**

Message ID	Message	Message Arguments	Notes
0x00000000	Request completed successfully.	None	None
0x00000001	Operation is not supported.	None	None
0x00000002	Failed. No further information is available.	None	None
0x00000003	Operation cannot complete within specified timeout period.	None	The operation was not initiated because it cannot complete within timeout period.
0x00000004	One or more parameters specified are invalid.	None	
0x00000005	Requested access is not supported.	None	SharedDeviceManagementService.Sharedevice( ) return code 6.
0x00000006	Use of timeout parameter is not supported.	None	None
0x00000007	The target device is busy and cannot be re-assigned.	None	None
0x00000008	Timeout expired prior to completion of operation.	None	Example: RequestStateChange( ) return code 3
0x00000009	An internal software error has occurred.	None	
0x0000000A	The target is busy and its state cannot be changed.	None	
0x0000000B	The target cannot transition to the requested state from its current state.	None	
0x0000000C	The target already exists.	None	
0x0000000D	The switch is in use, and the port mappings cannot be changed.	None	Example: AssignPorts( ) return code 3
0x0000000E	The specified ports are not mapped and cannot be unmapped.	None	Example: AssignPorts( ) return code 4
0x0000000F	The selected configuration is already active. Use the force option to re-apply it.	None	This error is appropriate when a client attempts to apply a configuration that is already active and to the mapping requires the force option to reapply the configuration.
0x00000010	The target association cannot be created between the specified targets.	None	This error is appropriate when a client attempts to create an association between two targets.

Message ID	Message	Message Arguments	Notes
0x00000011	A property value is incorrectly formatted.	None	This error is appropriate when the format used for a property value is incorrect.

8 SM CLP-to-CIM Common Verb Mappings

This section defines the mappings for verbs that are supported across all mappings.

8.1.1 cd

This section describes how to implement the cd verb irrespective of the current default target or the resultant target of a CLP command. Implementations must support the use of the cd verb for any command target.

8.1.1.1 Using cd without a Command Target Term

This command form corresponds to the cd verb specified without a command target term, in which the only behavior is to echo back the current default target.

Command Form: `cd`

CIM Requirements: `<CIM_CLPProtocolEndpoint.CurrentDefaultTarget>`

Behavior Requirements:

```
Pseudo code:
     $session = &smGetSession();
     //echo $session.CurrentDefaultTarget
     &smCommandCompleted($job);
```

8.1.1.2 Using cd with a Command Target Term

This command form corresponds to the cd verb specified with a command line term that is resolved to an absolute address.

Command Form: `cd <CIM_ManagedElement` *`single object`*`>`

CIM Requirements:

Behavior Requirements:

```
#address contains the UFiP that results from evaluating the command target term.

//this will generate an error if the address can't be resolved to an object path
$instance-> = &smGetObjectPath(#address);
$session = &smGetSession();
$session.CurrentDefaultTarget = $instance->;

#Error = &smOpModifyInstance ( $session, {"CurrentDefaultTarget"} );
if (0 != #Error.code) {
  &smProcessOpError (#Error);
  //includes smEnd;
}
//echo $session.CurrentDefaultTarget
&smCommandCompleted($job);
```

8.1.2 exit

This section describes how to implement the exit verb. The behavior of the exit verb is unaffected by the Current Default Target.

8.1.2.1 exit

This command form corresponds to all uses of the exit verb.

Command Form: `exit`

CIM Requirements:

Behavior Requirements:

```
   $Session = &smGetSession();
   &smDeleteInstance($Session.getObjectPath());
   &smEnd;
```

8.1.3 help

This section describes how to implement the help verb. The behavior of the help verb is not affected by the current default target.

8.1.3.1 help

This command form applies to all uses of the help verb.

Command Form: `help`

CIM Requirements:

Behavior Requirements:

```
//display help, behavior is implementation specific
```

```
    &smCommandCompleted($job);
```

8.1.4 version

This section describes how to implement the version verb. The behavior of the version verb is not affected by the Current Default Target.

8.1.4.1 version

This command form corresponds to all uses of the version verb.

Command Form: `version`

CIM Requirements:

Behavior Requirements:

```
$session = &smGetSession();

//1 Find the Service for the session
#Error = &smOpAssociatorNamess(
        $session->,
       "CIM_ProvidesEndpoint",
       "CIM_ProtocolService",
       NULL,
       NULL,
       NULL,
       $SvcInstancePaths[])
if (0 != #Error.code)
{
       &smProcessOpError (#Error);
       //includes smEnd;
}
//2 Find the capabilities for the Service
#Error = &smOpAssociators(
        $$SvcInstancePaths->[0],
       "CIM_ElementCapabilities",
       "CIM_CLPCapabilities",
       NULL,
       NULL,
       NULL,
       $CapInstances[])
if (0 != #Error.code)
{
       &smProcessOpError (#Error);
       //includes smEnd;
}

$cap = $CapInstances[0];
//echo $cap.CLPVersions[]
//echo $cap.SMMEAddressVersions[];
&smCommandCompleted($job);
```

ANNEX A
(informative)

Conventions

A.1 Notation

The following notations are used:

<*string*> Italicized text enclosed in angle brackets indicates that a substitution is to be made in the text where this string appears.

CIM_Classname, *CIM_Classname*

The full CIM class name is used to reference a CIM class (including the prefix “CIM_”). If the class is an abstract class, the entire class name string appears in italics.

<CIM_Classname> Angle brackets on either side of a CIM class name represent a target address term that resolves into one or more instances of the particular CIM class that is referenced. When this target address form is used, the command form is valid for single or multiple instances of the referenced class.

A.2 Pseudo-code

The command mappings in this specification are documented using pseudo-code to define the normalized behavior of the command execution. The pseudo-code sections are not intended to be compiled but are intended to be interpreted according to the particular CIM server interface used in the SM CLP implementation.

A.2.1 Pseudo-code Syntax

The pseudo-code conventions utilized in this document extend the Recipe Conventions that are defined in *Storage Management Initiative Specification*, section 7.6.

A.2.2 Pseudo-code Conventions

The pseudo-code used to define the command mapping behaviors uses the following conventions:

- `smEnd` indicates the end of the command mapping sequence for a single command mapping algorithm.
- When a function signature includes the assignment of a value to a parameter, the value assigned is the default value for the parameter, if the parameter is not specified when the method is called.

A.2.3 <ERROR> Data Type

The <ERROR> data type is patterned after the ERROR element returned within a METHODRESPONSE or IMETHODRESPONSE, as described in DSP0201, section 3.2.6.12. It contains the fields listed in Table A.1.

Table A.1 – ERROR Data Type Fields

Field	Notes
code	Code Attribute of <ERROR> element. A value of 0 indicates that the operation was successful. All other values are per the table in section 2.3.1.3 of DSP0200.
description	Description attribute of <ERROR> element
$error[]	Possibly null array of CIM_Error instances returned by operation

A.2.4 CommandStatus Data Type

The CommandStatus data type represents a Command Status construct as defined in the DSP0214. Table A.2 lists the fields in the data type and provides references to the relevant text within the *Server Management Command Line Protocol Specification*.

Table A.2 – CommandStatus Data Type Fields

Field	Notes
status	See DSP0214, Table 4.
status_tag	See DSP0214, Table 4.
job_id	See DSP0214, Table 8.
Errtype	See DSP0214, Table 8.
Errtype_desc	See DSP0214, Table 8.
Cimstat	See DSP0214, Table 8.
Cimstat_desc	See DSP0214, Table 8.
Severity	See DSP0214, Table 8.
Severity_desc	See DSP0214, Table 8.
Probcause	See DSP0214, Table 8.
Probcause_desc	See DSP0214, Table 8.
Recmdaction	See DSP0214, Table 8.
Errsource	See DSP0214, Table 8.
Errsourceform	See DSP0214, Table 8.
Errsourceform_desc	See DSP0214, Table 8.
Messages	One or more <Message> instances. See section A.2.5 for a definition of the structure.

At most, one instance of the CommandStatus data type is in scope for an operation. This instance is universally addressed as <CommandStatus>. Fields in the data type are addressed using standard dot notation.

For example, the `status` field is addressed as follows:

```
<CommandStatus>.status
```

See section 7.1.3 for example usage.

A.2.5 <Message> Data Type

The <Message> data type groups the set of CLP Command Status elements that comprise a single message. This data type is not intended to be used independently of the <CommandStatus> data type. Table A.3 lists the fields in the data type and provides references to the relevant text within the *Server Management Command Line Protocol Specification*.

Table A.3 – Message Data Type Fields

Field	Notes
message	See DSP0214, Table 6.
message_id	See DSP0214, Table 6.
message_arg	An array of message arguments. Each index corresponds to an occurrence of the message_arg keyword in the CLP Command Status. See DSP0214, Table 6.
Owningentity	See DSP0214, Table 6.

ANNEX B
(informative)

Per-Profile Mappings

The SM CLP-to-CIM command mappings for classes identified in a CIM Profile are documented in a separate per-profile SM CLP-to-CIM command mapping specification.

B.1 Preconditions and Assumptions for Per-Profile SM CLP-to-CIM Command Mapping Behaviors

Each of the command mappings in the per-profile command mapping specifications document the execution of a single SM CLP command issued to an implementation. The “Behavior Requirements” subsections contain the pseudo-code representation of the algorithm to be implemented to execute each command in a conformant manner.

B.1.1 Preconditions

– Job instance

  The implementation will create an instance of a CIM_Job to represent and track the execution of the command. The instance is stored in the variable #jobId. Any CIM_Error instances created are to be associated with this CIM_Job instance.

B.1.2 Assumptions

– Synchronous execution

  The algorithms documented in the “Behavior Requirements” subsections are assumed to be executed synchronously by the implementation. Any interruptions or parallel processing must be managed by the implementation. The resulting implementation must return command results as if the command execution were performed uninterrupted.

– Output stream

  Because the algorithms are assumed to be executed synchronously, the algorithms contain statements that indicate where output elements are to be generated sequentially into the output. If any buffering, caching, storing, or reorganization of the output data is performed, it is the responsibility of the implementation to produce the output in a conformant form.

A SM CLP-to-CIM command mapping specification specifies the level of support for each command that implementations must provide for CIM classes identified by a CIM profile. SM CLP command mappings specify the detailed description of command behavior when applied to instances of a specific CIM class, including expected state change and property value change behaviors. For each SM CLP command, the per-profile command mapping defines its “Support Requirement” by selecting one of the following levels of support: “shall”, “should”, “may”, or “shall not”.

For each SM CLP command that has a “Support Requirement” of “shall”, “should”, or “may”, the per-profile command mapping provides a “Description and Usage” statement and one or more Command Forms with corresponding CIM and Behavior Requirements. The “Description and Usage” statement describes, in general for the CIM class, how the user uses the command to manipulate a target instance of the class. Below the “Description and Usage” statement, the specification defines one or more command mappings, using the requirements specified in this *SM CLP-to-CIM Common Mapping Specification*.

Following each verb section header, each “Command Form” subsection denotes a particular form of the command that is specified by the *SM CLP-to-CIM Common Mapping Specification*. Each subsection describes a particular use of the verb, any required options and option arguments, target object instances, and property names and values specific to that use of the command verb.

For each command form, two types of specification are given:

- The “CIM Requirements” section specifies the particular CIM classes, properties, and methods that are mapped to the command in some manner. In most cases, this mapping is direct. In some cases, the mapping is indirect and may involve associations to related instances of other classes or multiple references into the same class.

- The “Behavior Requirements” section specifies any detailed mapping requirements that must be followed by the implementation. The “Behavior Requirements” are documented using the pseudo-code syntax defined in section A.2.

Commands may act on either a specific instance or many instances. In the “Command Form” section, the notation “`<CIM_classname single instance>`” is used to denote that a single instance target is used in the command form. If the command is supported for multiple instances of the class, a second command mapping item is included, specifying the behavior when the target address term resolves to multiple instances of the class. The notation “`<CIM_classname multiple instances>`” indicates that the command target resolves into multiple instances in this command form.

ANNEX C
(informative)

Change Log

Version	Date	Author	Description
1.0.0	4/23/2009		DMTF Standard Release

Bibliography

DMTF DSP0217, *SMASH Implementation Requirements*, 1.0.0,
http://www.dmtf.org/standards/published_documents/DSP0217_1.0.0.pdf

DMTF DSP2001, *Systems Management Architecture for Server Hardware (SMASH) Command Line Protocol (CLP) Architecture White Paper, 1.0.0,*
http://www.dmtf.org/standards/published_documents/DSP2001_1.0.1.pdf