SoftwareManagement:1

Service Template Version 1.01

For UPnP Version 1.0

Status: Standardized DCP (SDCP)

Date: July 20, 2010

Authors	Company
William Lupton	2Wire
Andre Bottaro	France Telecom Group
Francois-Gaël Ottogalli	France Telecom Group
Levent Gurgen	France Telecom Group
Nicolas Chabanoles	France Telecom Group
Jooyeol Lee	Samsung Electronics
Kiran Vedula (Editor)	Samsung Electronics
Davide Moreo	Telecom Italia

* Note: The UPnP Forum in no way guarantees the accuracy or completeness of this author list and in no way implies any rights for or support from those members listed. This list is not the specifications' contributor list that is kept on the UPnP Forum's website.

Contents

List of Tables

List of Figures

1 Overview and Scope

This service definition is compliant with the UPnP Device Architecture version 1.0 [UDA1.0]. It defines a service type referred to herein as *SoftwareManagement:1* service.

1.1 Introduction

Many devices in the home network (for example, TV, Camera, Media Server etc.) have embedded software or firmware that needs to be managed. The *SoftwareManagement:1* service is intended to provide control points with the ability to manage this embedded software or firmware.

The SoftwareManagement service enables a control point to:

- Install software entities on a targeted Execution Environment.
- Uninstall software entities on a targeted Execution Environment.
- Start software entities on a targeted Execution Environment.
- Stop software entities on a targeted Execution Environment.
- Update device firmware.

The SoftwareManagement service does not:

- Mandate the protocol used for downloading the DUs (Deployment Unit)
- Provide details of any particular Execution Environment

This specification frequently uses the term *Parent Device*. This refers to UPnP device/service sub-tree whose root is the UPnP device that contains the *SoftwareManagement:1* service instance. UPnP actions or other operations on a *Parent Device* SHOULD apply to all levels of this sub-tree, but SHOULD NOT apply to an embedded device that itself contains a *SoftwareManagement:1* service instance.

1.2 References

This section lists the references used in the UPnP DM specifications and includes the tag inside square brackets that is used for each such reference:

[BMS]	*UPnP BasicManagement:1 Service Document*, UPnP Forum, July 20, 2010. Available at: www.upnp.org/specs/dm/UPnP-dm-BasicManagement-v1.0-Service.pdf
[CMS]	*UPnP ConfigurationManagement:1 Service Document*, UPnP Forum, July 20, 2010. Available at: http://www.upnp.org/specs/dm/UPnP-dm-ConfigurationManagement-v1.0-Service.pdf
[CMS-XSD]	*XML Schema for ConfigurationManagement: 1*, UPnP Forum, July 5, 2010, http://www.upnp.org/schemas/dm/cms-v1.xsd
[DEVICE]	*UPnP ManageableDevice:1 Device Document*, UPnP Forum, July 20, 2010. Available at: http://www.upnp.org/specs/dm/UPnP-dm-ManageableDevice-v1.0-Device.pdf
[MIDP]	*Mobile Information Device Profile for Java™ 2 Micro Edition Version 2.0*, Java Community Process, November 2002.
[OSGi]	*OSGi Core Specification and Service Compendium*, Release 4 Version 4.1, OSGi Alliance, April 2007.

[RF07] *A Survey of Unix Init Schemes*, Yvan Royon, Stéphane Frénot, Technical Report, Inria RT-0338, June 2007.

[REQLEV] *RFC 2119*, *Key words for use in RFCs to Indicate Requirement Levels*, S.Bradner, 1997. Available at: http://www.ietf.org/rfc/rfc2119.txt

[SCOMO] *Software Component Management Object*, Open Mobile Alliance (OMA), Draft Version 1.0, June 2008.

[UDA1.0] *UPnP Device Architecture, version1.0*, UPnP Forum, July 20, 2006. Available at: http://www.upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.0.pdf

[XML] *Extensible Markup Language (XML) 1.0 (Third Edition)*, W3C, February 2004, http://www.w3.org/TR/2004/REC-xml-20040204

[XML-NS] *The "xml:" Namespace*, W3C, April 2006, http://www.w3.org/XML/1998/namespace

[XML-NMSP] *Namespaces in XML*, W3C, August 2006, http://www.w3.org/TR/REC-xml-names

[XML-SCHEMA-1] *XML Schema Part 1: Structures Second Edition*, W3C, October 2004, http://www.w3.org/TR/xmlschema-1

[XML-SCHEMA-2] *XML Schema Part 2: Datatypes Second Edition*, W3C, October 2004, http://www.w3.org/TR/xmlschema-2

1.3 Glossary

BMS	*BasicManagement Service*
CMS	*ConfigurationManagement Service*
CSD	*Configuration State Diagram*
CSV	*Comma-Separated Value*
DU	*Deployment Unit*
DUID	*Deployment Unit identifier*
EE	*Execution Environment*
EU	*Execution Unit*
EUID	*Execution Unit identifier*
RSD	*Running State Diagram*
SMS	*SoftwareManagement Service*

1.4 Notation

- In this document, features are described as Required, Recommended, or Optional as follows:

  The key words "MUST," "MUST NOT," "REQUIRED," "SHALL," "SHALL NOT," "SHOULD," "SHOULD NOT," "RECOMMENDED," "MAY," and "OPTIONAL" in this specification are to be interpreted as described in [REQLEV].

  In addition, the following keywords are used in this specification:

  PROHIBITED – The definition or behavior is an absolute prohibition of this specification. Opposite of REQUIRED.

  CONDITIONALLY REQUIRED – The definition or behavior depends on a condition. If the specified condition is met, then the definition or behavior is REQUIRED, otherwise it is PROHIBITED.

  CONDITIONALLY OPTIONAL – The definition or behavior depends on a condition. If the specified condition is met, then the definition or behavior is OPTIONAL, otherwise it is PROHIBITED.

  These keywords are thus capitalized when used to unambiguously specify requirements over protocol and application features and behavior that affect the interoperability and security of implementations. When these words are not capitalized, they are meant in their natural-language sense.

- Strings that are to be taken literally are enclosed in "double quotes."
- Words that are emphasized are printed in *italic*.
- Data model names and values, and literal XML, are printed using the `data` character style.
- Keywords that are defined by the UPnP DM Working Committee are printed using the *forum* character style.
- Keywords that are defined by the UPnP Device Architecture are printed using the **arch** character style.
- A double colon delimiter, "::", signifies a hierarchical parent-child (parent::child) relationship between the two objects separated by the double colon. This delimiter is used in multiple contexts, for example: Service::Action(), Action()::Argument.

1.4.1 Data Types

This specification uses data type definitions from two different sources. The UPnP Device Architecture defined data types are used to define state variable and action argument data types UDA. The XML Schema namespace is used to define XML-valued action arguments [XML-SCHEMA-2] (including [CMS] data model parameter values).

For UPnP Device Architecture defined Boolean data types, it is strongly RECOMMENDED to use the value "**0**" for false, and the value "**1**" for true. However, when used as input arguments, the values "**false**", "**no**", "**true**", "**yes**" may also be encountered and MUST be accepted. Nevertheless, it is strongly RECOMMENDED that all state variables and output arguments be represented as "**0**" and "**1**".

For XML Schema defined Boolean data types, it is strongly RECOMMENDED to use the value "*0*" for false, and the value "*1*" for true. However, when used within input arguments, the values "*false*", "*true*" may also be encountered and MUST be accepted. Nevertheless, it is strongly RECOMMENDED that all XML Boolean values be represented as "*0*" and "*1*".

XML elements that are of type `xsd:anySimpleType` (for example [CMS] data model parameter values) MUST include an `xsi:type` attribute that indicates the actual data type of the element value. This is a SOAP requirement.

1.4.2 Strings Embedded in Other Strings

Some string variables, arguments and other XML elements and attributes (including [CMS] data model parameter values) described in this document contain substrings that MUST be independently identifiable and extractable for other processing. This requires the definition of appropriate substring delimiters and an escaping mechanism so that these delimiters can also appear as ordinary characters in the string and/or its independent substrings.

This document uses such embedded strings in Comma Separated Value (CSV) lists (see section 1.5.1). Escaping conventions use the backslash character, "\" (character code U+005C), as follows:

a) Backslash ("\") is represented as "\\".

b) Comma (",") is represented as "\," in individual substring entries.

c) Double quote (""") is not escaped.

This document also uses such embedded strings to represent XML documents (see section 1.5.2). Escaping conventions use XML entity references as specified in [XML] Section 2.4. For example:

a) Ampersand ("&") is represented as "`&amp;`" or via a numeric character reference.

b) Left angle bracket ("<") is represented as "`&lt;`" or via a numeric character reference.

c) Right angle bracket (">") usually doesn't have to be escaped, but often is, in which case it is represented as "`&gt;`" or via a numeric character reference.

1.5 Derived Data Types

This section defines a derived data type that is represented as a string data type with special syntax. This specification uses string data type definitions that originate from two different sources. The UPnP Device Architecture defined **string** data type is used to define state variable and action argument string data types. The XML Schema namespace is used to define `xsd:string` data types. The following definition applies to both string data types.

1.5.1 Comma Separated Value (CSV) Lists

The UPnP DM services use state variables, action arguments and other XML elements and attributes that represent lists – or one-dimensional arrays – of values. UDA does not provide for either an array type or a list type, so a list type is defined here. Lists MAY either be homogeneous (all values are the same type) or heterogeneous (values of different types are allowed). Lists MAY also consist of repeated occurrences of homogeneous or heterogeneous subsequences, all of which have the same syntax and semantics (same number of values, same value types and in the same order).

- The data type of a homogeneous list is **string** or `xsd:string` and denoted by CSV (x), where x is the type of the individual values.

- The data type of a heterogeneous list is also **string** or `xsd:string` and denoted by CSV (w, x [, y, z]), where w, x, y and z are the types of the individual values, and the square brackets indicate that y and z (and the preceding comma) are optional. If the number of values in the heterogeneous list is too large to show each type individually, that variable type is represented as CSV (*heterogeneous*), and the variable description includes additional information as to the expected sequence of values appearing in the list and their corresponding types. The data type of a repeated subsequence list is **string** or `xsd:string` and denoted by CSV ({w, x, y, z}), where w, x, y and z are the types of the individual values in the subsequence and the subsequence MAY be repeated zero or more times (in this case none of the values are optional).

The individual value types are specified as UDA data types or **A_ARG_TYPE** data types for **string** lists, and as [XML-SCHEMA-2] data types for `xsd:string` lists.

- A list is represented as a **string** type (for state variables and action arguments) or `xsd:string` type (within other XML elements and attributes).

- Commas separate values within a list.

- Integer values are represented in CSVs with the same syntax as the integer data type specified in UDA (that is: optional leading sign, optional leading zeroes, numeric ASCII).

- Boolean values are represented in state variable and action argument CSVs as either "**0**" for false or "**1**" for true. These values are a subset of the defined Boolean data type values specified in UDA: **0**, **false**, **no**, **1**, **true**, **yes**.

- Boolean values are represented in other XML element CSVs as either "`0`" for false or "`1`" for true. These values are a subset of the defined Boolean data type values specified in [XML-SCHEMA-2]: `0`, `false`, `1`, `true`.

- Escaping conventions for the comma and backslash characters are defined in section 1.4.2.

- The number of values in a list is the number of unescaped commas, plus one. The one exception to this rule is that an empty string represents an empty list. This means that there is no way to represent a list consisting of a single empty string value.

- White space before, after, or interior to any numeric data type is not allowed.

- White space before, after, or interior to any other data type is part of the value.

Table 1-1: CSV Examples

Type refinement of string	Value	Comments
CSV (**string**)	"first,second"	List of 2 strings used as state variable or action argument value.
CSV (`xsd:string`)	"first,second"	List of 2 strings used within an XML element

Type refinement of string	Value	Comments
CSV (`xsd:token`)	"first, second "	List of 2 strings used within an XML element. Each element is of type `xsd:token` so, even though the second value is " second " and has leading and trailing spaces, the value seen by the application will be "second" because `xsd:token` collapses whitespace.
CSV (**string**, **date-Time.tz** [, **string**])	"Warning,2009-07-07T13:22:41, third\,value"	List of **string**, **dateTime.tz** and (optional) **string** used as state variable or action argument value. Note the leading space and escaped comma in the third value, which is " third,value".
CSV (**string**, **date-Time.tz** [, **string**])	"Warning,2009-07-07T13:22:41,"	As above but third value is empty.
CSV (**string**, **date-Time.tz** [, **string**])	"Warning,2009-07-07T13:22:41"	As above but third value is omitted.
CSV (*A_ARG_TYPE_-Host*)	"grumpy,sleepy"	List of data items used as action argument value, each of which obeys the rules governing *A_ARG_TYPE_Host*. Any comma or backslash characters within a data item would have been escaped.
CSV (**i4**)	"1, 2"	Illegal CSV. White space is not allowed as part of an integer value.
CSV (**string**)	"a,,c,"	List of 4 strings "a", "", "c" and "".
CSV (**string**)	""	Empty list. It is not possible to create a list containing a single empty string.

1.5.2 Embedded XML Documents

An XML document is a string that represents a valid XML 1.0 document according to a specific schema. Every occurrence of the phrase "*XML Document*" is italicized and preceded by the document's root element name (also italicized), as listed in column 3, "Valid Root Element(s)" of Table 1-3, "Schema-related Information". For example, the phrase *SupportedDataModels XML Document* refers to a valid XML 1.0 document according to the CMS schema [CMS-XSD]. Such a document comprises a single `<SupportedDataModels …>` root element, optionally preceded by the XML declaration `<?xml version="1.0" …?>`.

This string will therefore be of one of the following two forms:

"`<SupportedDataModels …>…</SupportedDataModels>`"

or

"`<?xml …?><SupportedDataModels …>…</SupportedDataModels>`"

Escaping conventions for the ampersand, left angle bracket and right angle bracket characters are defined in section 1.4.2.

1.6 Management of XML Namespaces in Standardized DCPs

UPnP specifications make extensive use of XML namespaces. This allows separate DCPs, and even separate components of an individual DCP, to be designed independently and still avoid name collisions when they share XML documents. Every name in an XML document belongs to exactly one namespace. In documents, XML names appear in one of two forms: qualified or unqualified. An unqualified name (or no-colon-name) contains no colon ("`:`") characters. An unqualified name belongs to the document's default namespace. A qualified name is two no-colon-names separated by one colon character. The no-colon-name before the colon is the qualified name's namespace prefix, the no-colon-name after the colon is the qualified name's "local" name (meaning local to the namespace identified by the namespace prefix). Similarly, the unqualified name is a local name in the default namespace.

The formal name of a namespace is a URI. The namespace prefix used in an XML document is not the name of the namespace. The namespace name is, or should be, globally unique. It has a single definition that is accessible to anyone who uses the namespace. It has the same meaning anywhere that it is used, both inside and outside XML documents. The namespace prefix, however, in formal XML usage, is defined only in an XML document. It must be locally unique to the document. Any valid XML no-colon-name may be used. And, in formal XML usage, no two XML documents are ever required to use the same namespace prefix to refer to the same namespace. The creation and use of the namespace prefix was standardized by the W3C XML Committee in [XML-NMSP] strictly as a convenient local shorthand replacement for the full URI name of a namespace in individual documents.

All of the namespaces used in this specification are listed in the Tables "Namespace Definitions" and "Schema-related Information". For each such namespace, Table 1-2, "Namespace Definitions" gives a brief description of it, its name (a URI) and its defined "standard" prefix name. Some namespaces included in these tables are not directly used or referenced in this document. They are included for completeness to accommodate those situations where this specification is used in conjunction with other UPnP specifications to construct a complete system of devices and services. The individual specifications in such collections all use the same standard prefix. The standard prefixes are also used in Table 1-3, "Schema-related Information", to cross-reference additional namespace information. This second table includes each namespace's valid XML document root element(s) (if any), its schema file name, versioning information (to be discussed in more detail below), and a link to the entry in Section 1.2, "References" for its associated schema.

The normative definitions for these namespaces are the documents referenced in Table 1-3. The schemas are designed to support these definitions for both human understanding and as test tools. However, limitations of the XML Schema language itself make it difficult for the UPnP-defined schemas to accurately represent all details of the namespace definitions. As a result, the schemas will validate many XML documents that are not valid according to the specifications.

Table 1-2: Namespace Definitions

Standard Name-space Prefix	Namespace Name	Namespace Description	Normative Definition Document Reference
	DM Working Committee defined namespaces		
`cms`	urn:schemas-upnp-org:dm:cms	CMS data structures	[CMS]
`bmsnsl`	urn:schemas-upnp-org:dm:bms:nsl	BMS NSLookupResult	[BMS]
	Externally defined namespaces		
`xsd`	http://www.w3.org/2001/XMLSchema	XML Schema Language 1.0	[XML-SCHEMA-1] [XML-SCHEMA-2]
`xsi`	http://www.w3.org/2001/XMLSchema-instance	XML Schema Instance Document schema	Sections 2.6 & 3.2.7 of [XML-SCHEMA-1]
`xml`	http://www.w3.org/XML/1998/namespace	The "xml:" Namespace	[XML-NS]

Table 1-3: Schema-related Information

Standard Name-space Prefix	Relative URI and File Name[1] • Form 1, 2, 3	Valid Root Element(s)	Schema Reference
	DM Working Committee defined namespaces		

`cms`	• cms-v*n*-*yyyymmdd*.xsd
	• cms-v*n*.xsd
	• cms.xsd
		`<StructurePathList>`
		`<ParameterPathList>`
		`<ParameterAttributeList>`
		`<InstanceValueList>`
		`<SupportedDataModels>`
		`<InstancePathList>`
		`<ContentPathList>`
		`<AttributePathList>`
			[CMS]

`bmsnsl`	• bmsnsl-v*n*-*yyyymmdd*.xsd • bmsnsl-v*n*.xsd • bmsnsl.xsd	`<NSLookupResult>`	[BMS]

[1] Absolute URIs are generated by prefixing the relative URIs with "http://www.upnp.org/schemas/dm/".

1.6.1 Namespace Names, Namespace Versioning and Schema Versioning

The UPnP DM service specifications define several data structures (such as state variables and action arguments) whose format is an XML instance document that must comply with one or more specific XML namespaces. Each namespace is uniquely identified by an assigned namespace name. The namespaces that are defined by the DM Working Committee MUST be named by a URN. See Table 1-2 "Namespace Definitions" for a current list of namespace names. Additionally, each namespace corresponds to an XML schema document that provides a machine-readable representation of the associated namespace to enable automated validation of the XML (state variable or action parameter) instance documents.

Within an XML schema and XML instance document, the name of each corresponding namespace appears as the value of an `xmlns` attribute within the root element. Each `xmlns` attribute also includes a namespace prefix that is associated with that namespace in order to disambiguate (a.k.a. qualify) element and attribute names that are defined within different namespaces. The schemas that correspond to the listed namespaces are identified by URI values that are listed in the `schemaLocation` attribute also within the root element. (See Section 1.6.2)

In order to enable both forward and backward compatibility, namespace names are permanently assigned and MUST NOT change even when a new version of a specification changes the definition of a namespace. However, all changes to a namespace definition MUST be backward-compatible. In other words, the updated definition of a namespace MUST NOT invalidate any XML documents that comply with an earlier definition of that same namespace. This means, for example, that a namespace MUST NOT be changed so that a new element or attribute is required. Although namespace names MUST NOT change, namespaces still have version numbers that reflect a specific set of definitional changes. Each time the definition of a namespace is changed, the namespace's version number is incremented by one.

Each time a new namespace version is created, a new XML schema document (.xsd) is created and published so that the new namespace definition is represented in a machine-readable form. Since an XML schema document is just a representation of a namespace definition, translation errors can occur. Therefore, it is sometime necessary to re-release a published schema in order to correct typos or other namespace representation errors. In order to easily identify the potential multiplicity of schema releases for the same namespace, the URI of each released schema MUST conform to the following format (called Form 1):

Form 1: "http://www.upnp.org/schemas/dm/" ***schema-root-name*** "-v" ***ver*** "-" ***yyyymmdd***

where

- ***schema-root-name*** is the name of the root element of the namespace that this schema represents.
- ***ver*** corresponds to the version number of the namespace that is represented by the schema.
- ***yyyymmdd*** is the year, month and day (in the Gregorian calendar) that this schema was released.

Table 1-3 "Schema-related Information" identifies the URI formats for each of the namespaces that are currently defined by the UPnP DM Working Committee.

As an example, the original schema URI for the "`cms`" namespace might be "http://www.upnp.org/schemas/dm/cms-v1-20091231.xsd". If the UPnP DM service specifications were subsequently updated in the year 2010, the URI for the updated version of the "`cms`" namespace might be "http://www.upnp.org/schemas/dm/cms-v2-20100906.xsd".

In addition to the dated schema URIs that are associated with each namespace, each namespace also has a set of undated schema URIs. These undated schema URIs have two distinct formats with slightly different meanings:

Form 2: "http://www.upnp.org/schemas/dm/" ***schema-root-name*** "-v" ***ver***

Form 3: "http://www.upnp.org/schemas/dm/" ***schema-root-name***

Form 2 of the undated schema URI is always linked to the most recent release of the schema that represents the version of the namespace indicated by ***ver***. For example, the undated URI ".../dm/cms-v2.xsd" is linked to the most recent schema release of version 2 of the "`cms`" namespace. Therefore, on September 06, 2010 (20100906), the undated schema URI might be linked to the schema that is otherwise known as ".../dm/cms-v2-20100906.xsd". Furthermore, if the schema for version 2 of the "`cms`" namespace was ever re-released, for example to fix a typo in the 20100906 schema, then the same undated schema URI (".../dm/cms-v2.xsd") would automatically be updated to link to the updated version 2 schema for the "`cms`" namespace.

Form 3 of the undated schema URI is always linked to the most recent release of the schema that represents the highest version of the namespace that has been published. For example, on December 31, 2009 (20091231), the undated schema URI ".../dm/cms.xsd" might be linked to the schema that is otherwise known as ".../dm/cms-v1-20091231.xsd". However, on September 06, 2010 (20100906), that same undated schema URI might be linked to the schema that is otherwise known as ".../dm/cms-v2-20100906.xsd". When referencing a schema URI within an XML instance document or a referencing XML schema document, the following usage rules apply:

- All instance documents, whether generated by a service or a control point, MUST use Form 3.
- All UPnP DM published schemas that reference other UPnP DM schemas MUST also use Form 3.

Within an XML instance document, the definition for the `schemaLocation` attribute comes from the XML Schema namespace "http://www.w3.org/2002/XMLSchema-instance". A single occurrence of the attribute can declare the location of one or more schemas. The `schemaLocation` attribute value consists of a whitespace separated list of values that is interpreted as a namespace name followed by its schema location URL. This pair-sequence is repeated as necessary for the schemas that need to be located for this instance document.

In addition to the schema URI naming and usage rules described above, each released schema MUST contain a `version` attribute in the `<schema>` root element. Its value MUST correspond to the format:

> ***ver*** "-" ***yyyymmdd*** where ***ver*** and ***yyyymmdd*** are described above.

The `version` attribute provides self-identification of the namespace version and release date of the schema itself. For example, within the original schema released for the "`cms`" namespace (.../cms-v1-20091231.xsd), the `<schema>` root element might contain the following attribute: `version="1-20091231"`.

1.6.2 Namespace Usage Examples

The `schemaLocation` attribute for XML instance documents comes from the XML Schema instance namespace "http:://www.w3.org/2001/XMLSchema-instance". A single occurrence of the attribute can declare the location of one or more schemas. The `schemaLocation` attribute value consists of a whitespace separated list of values: namespace name followed by its schema location URL. This pair-sequence is repeated as necessary for the schemas that need to be located for this instance document.

Example 1:

Sample CMS XML Instance Document. Note that the references to the UPnP DM schemas do not contain any version or release date information. In other words, the references follow Form 3 from above. Consequently, this example is valid for all releases of the UPnP DM service specifications.

```
<?xml version="1.0" encoding="UTF-8"?>
<cms:ParameterValueList
 xmlns:cms="urn:schemas-upnp-org:dm:cms"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="urn:schemas-upnp-org:dm:cms
```

```
                     http://www.upnp.org/schemas/dm/cms.xsd">
  <Parameter>
    <Path>...</Path>
    <Value>...</Value>
  </Parameter>
  ...
</cms:ParameterValueList>
```

1.7 Vendor-defined Extensions

Whenever vendors create additional vendor-defined state variables, actions or other XML elements and attributes, their assigned names and XML representation MUST follow the naming conventions and XML rules as specified in UDA, Section 2.5, “Description: Non-standard vendor extensions”.

2 Service Modeling Definitions

2.1 Service Type

The following service type identifies a service that is compliant with this template:

> **urn:schemas-upnp-org:service:***SoftwareManagement:1*.

2.2 Key Concepts

This section is added to describe some normative requirements for the *SoftwareManagement* Service.

2.2.1 Software entities

There are two software entities defined by SoftwareManagement service which are defined as follows:

- **Deployment Unit (DU):** A DU is an entity which packages resources such as library files, executable files, configuration files, jar files, bundles or assemblies. This entity can be individually installed, uninstalled and updated. Each DU is identified using a unique identifier DUID.
- **Execution Unit (EU):** A EU is an entity such as a service, script, software component or MIDlet. It can individually be started or stopped. Once started an EU becomes active, e.g., provides services, until it is stopped. EUs only appear after the installation or update of DUs. Each EU is linked to a unique DU. A DU can embed multiple EUs. Each EU is identified using a unique identifier EUID.

Actions pertain to one of these two entities, e.g., *Install()*, *Update()* and *Uninstall()* pertain only to DUs. Each entity has a separate unique *Identifier* to identify it uniquely across a *Parent Device.*

2.2.2 Software Data Model

SMS defines an Optional extension to the UPnP DM data model (*Common Objects*). This extension is defined in `/UPnP/DM/Software/` data model sub-tree which contains details of DUs and EUs. Implementations which choose to provide additional details to the control point will do so by implementing the *Software Data Model* and a CMS to provide access to the data model. More details about the supported *Parameters* for the DU and EU are presented in Appendix B.

2.2.3 Lifecycle Management

2.2.3.1 Deployment Unit lifecycle management

A DU can be installed and uninstalled using the *Install()* and the *Uninstall()* actions respectively. If *Software Data Model* is implemented, the new DU state(a result of the execution of those actions) MAY be stored into the data model as a read-only *parameter* named (see Appendix B Software Data Model):

```
/UPnP/DM/Software/DU/#/State
```

The # represents an instance number of a DU in a table of known DUs.

The allowed values for that *Parameter* are (see Figure 2-1):

- *Installing*: represents a transitory state while the DU is being installed.
- *Unresolved*: represents a stable state in which the DU is installed but some of its dependencies are missing.
- *Installed*: represents a stable state in which the DU is currently installed and all the needed dependencies, if any, are resolved.
- *Uninstalling*: represents the transitory state while the DU is being uninstalled.
- *Uninstalled*: represents a stable state in which the DU has been uninstalled.

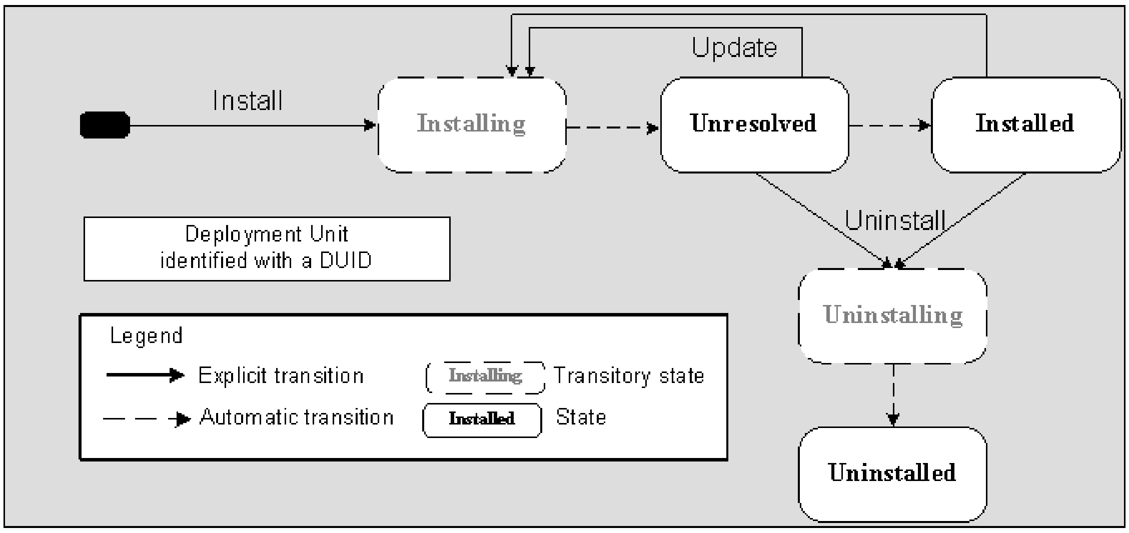


Figure 2-1: DU state diagram

Two types of transitions are depicted in Figure 2-1, i.e., explicit transitions and automatic transitions. Explicit transitions are triggered by the request of a software management action. Such a request can be made by invoking a SMS action or any external API. The remainder of this section only describes the explicit transitions triggered by the call of SMS actions.

The *Installing* state can be reached by calling the *Install()* or the *Update()* action.

The *Installed* state is reached as an automatic transition from the *Installing* or the *Unresolved* state when the installation succeeds.

The *Unresolved* state is reached as an automatic transition from the *Installing* state when the action completes but some dependencies are not currently resolved; i.e., this is not a success state post-installation.

The *Uninstalling* state can be reached by calling the *Uninstall()* action on a DU which is in *Unresolved* or *Installed* state.

The *Uninstalled* state can be reached as an automatic transition from the *Uninstalling* state when the *Uninstall ()* action completes.

2.2.3.2 Execution Unit lifecycle management

For an EU there is need for two distinct states called the *Requested State* and *Running State*.

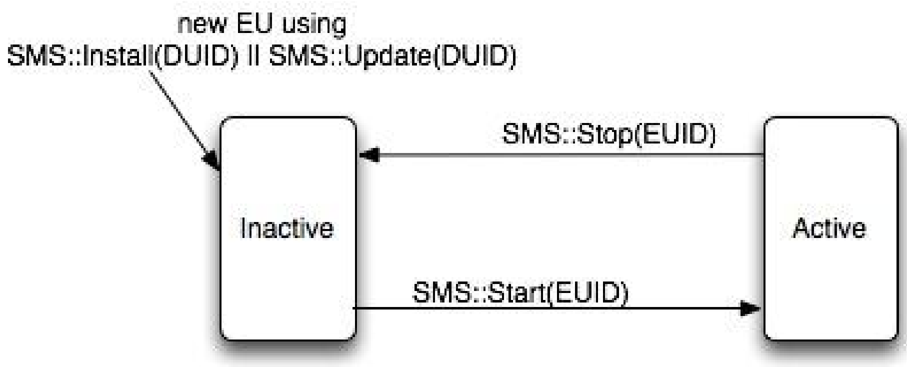
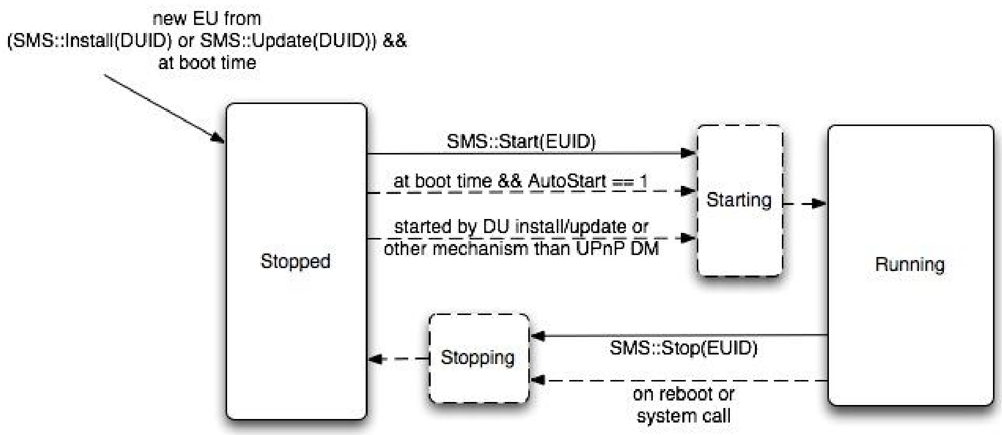


Figure 2-2: EUs' Requested State

When a control point requests an EU to *Start*, it will expect the EU to be *Active*. Similarly when a control point requests an EU to *Stop*, it will expect the EU to be *Inactive*. These two states "*Active*" and "*Inactive*" are called the *Requested states* from a control point point of view. But as per *Parent Device*, the actual EU state can be *Starting*, *Running, Stopping* and *Stopped*. These are called the *Running States*.



new EU from
(SMS::Install(DUID) or SMS::Update(DUID)) &&
at boot time
Stopped
SMS::Start(EUID)
at boot time && AutoStart == 1
started by DU install/update or
other mechanism than UPnP DM
Starting
Running
Stopping
SMS::Stop(EUID)
on reboot or
system call

Figure 2-3: EUs' Running State

The *Requested* state is defined according to what action is called on the EU. If *Software Data Model* is implemented, the EU's *Requested* state MAY be provided by the read-only *Parameter* named (see Appendix B Software Data Model):

`/UPnP/DM/Software/DU/#/EU/#/RequestedState`

The *Parameter* allowed values are:

- *Active*: The value is set to *Active* when the *Start()* action is called (see 0).
- *Inactive*: The value is set to *Inactive* when the *Stop()* action is called (see 0).

If the *Software Data Model* is implemented, the EU behaviour MAY be managed by two optional *Parameters* (see Appendix B Software Data Model):

`/UPnP/DM/Software/DU/#/EU/#/AutoStart`

`/UPnP/DM/Software/DU/#/EU/#/AutoRestart`

AutoStart: When set to *1* and once the EU is in Active state, this indicates that the EU MUST be automatically started when the *ParentDevice* boots. If *Software Data Model* is implemented, the *AutoStart Parameter* relies on the EE's ability `/UPnP/DM/Software/Capabilities/AbleToAutoStart` to autonomously start EUs at boot time. If *Software Data Model* is not implemented, the *AutoStart* takes a default value of 0.

AutoRestart: When set to *1* and once the EU is in Active state, this indicates that the EU will be restarted each time it is stopped by any other means than the *Stop()* action (e.g., a reboot). If the *Software Data Model* is implemeted, the *AutoRestart Parameter* relies on the EE's ablility `/UPnP/DM/Software/Capabilities/AbleToAutoRestart`to implement a mechanism such as a watchdog. *AutoRestart Parameter* has no effect at boot time. If *Software Data Model* is not implemented, the *AutoRetart* takes a default value of 0.

Both *Parameters* are only relevant when the *RequestedState* value is *Active*.

The *Running State* is defined according to what is currently happening on the targeted EE . If *Software Data Model* is implemented, the EU's *Running State* MAY be provided by the read-only *Parameter* named (see Appendix B Software Data Model):

`/UPnP/DM/Software/DU/#/EU/#/RunningState`

Running State allowed values are:

- *Stopped*: the EU is observed as stopped on the EE.
- *Stopping*: represents the transitory state while the EU is being stopped.
- *Running*: the EU is observed as running on the EE.
- *Starting*: represents the transitory state while the EU is being started.

The *AutoStart Parameter* MUST be effective only when the *Start()* action is called atleast once i.e. when the *Requested State* of an EU is *Active*. When the Requested State of EU is *Inactive*, the *AutoStart* MUST be ignored.

An Execution Unit (EU) comes into existence because a DU is installed. If the *Software Data Model* is implemented, the EU configuration MAY be maintained in the *Software Data Model* (see Appendix B Software Data Model). More details of the supported *Parameters* can be found in Appendix B.

2.2.4 Firmware

Some specific EE define the notion of firmware. A firmware corresponds to a basic piece of software including a bootloader; without the baseline firmware installed, the device might not function. Several firmware images could be present at the same time, but only one is used at boot time. This one is called the primary firmware. DUID and EUID 0 are reserved for identifying this primary firmware.

2.2.5 Asynchronous actions

Software management operations (e.g., the start of an EU) are expected to take a significant length of time to execute. In particular, it cannot be assumed that a given operation will complete within the 30 seconds allowed by the UDA. Therefore, every operation is done asynchronously: an action initiates the operation and an *OperationID* is immediately returned. Events associated to the *OperationIDs* state variable indicate the end of the operation to the service subscribers. Other related information SHOULD be notified in the same event message, i.e., the modified *DUIDs*, *EUIDs*, *ActiveEUIDs* lists.

2.2.6 Software entity dependency management

Software entities on a particular EE MAY have dependencies on each other; the ability for an EE to intelligently manage these dependencies is implementation (or EE) specific.

A control point can know about the dependency handling capability of a *Parent Device* in two ways:

- either by calling any action like *install()/Start()* and checking its response or
- by retrieving the value of the `/UPnP/DM/Software/Capabilities/AbleToHandleDUDependency` and `/UPnP/DM/Software/Capabilities/AbleToHandleEUDependency` *Parameters* in the *Software Data Model* (if it is supported).

SMS actions have a Boolean argument called HandleDependencies which a control point uses to indicate to the device if it wants the device to handle dependencies automatically or if the control point itself will take care of the dependency handling. If an implementation can not honor HandleDependencies request, it MUST return an error. If a device does not return any error for HandleDependencies, the device MUST honor the HandleDependencies request and accordingly MUST either handle dependencies on its own when HandleDependencies is *1* and MUST NOT handle dependencies on its own when HandleDependencies is *0*.

Alternatively a control point can check the dependency handling capability *Parameter* of a device in the *Software Data Model*. If the capability is *0* and a control point invokes an action with HandleDependencies argument equal to *1*, then the device MUST return an error. In other words if the capability is *0* then a control point cannot control the dependency handling of this device. In that case a control point MUST handle the dependencies on its own. But if the capability is *1* and a control point invokes an action with HandleDependencies as *0*, then the device, despite having the ability to handle dependencies, it MUST NOT try to handle dependencies on its own. If, however the capability is *1* and a control point invokes an action with HandleDependencies as *1*, then the device MUST try to handle dependencies.

All the actions that have “HandleDependencies” as anargument MUST comply with the above requirements. The exact details of what a dependency is and how and what are the requirements to handle the dependencies are implementation (or EE) specific. The DU dependencies and EU dependencies could have different semantics in different implementations.

If some dependencies of a DU are not satisfied for whatever reason, the DU MUST be assigned the *Unresolved* state (see Figure 2-1). If all the dependencies are successfully installed, the DU state is set to *Installed*. In both scenarios, all the EUs will be in the *Inactive* state.

2.3 State Variables

Table 2-1: State Variables

Variable Name	Req. or Opt.[1]	Data Type	Allowed Value	Default Value [2]	Eng. Units
OperationIDs	*R*	**string**	CSV(*A_ARG_TYPE_ID*) See section 2.3.1	""	
DUIDs	*R*	**string**	CSV(*A_ARG_TYPE_ID*) See section 2.3.2	""	
EUIDs	*R*	**string**	CSV(*A_ARG_TYPE_ID*) See section 2.3.3	""	
ActiveEUIDs	*R*	**string**	CSV(*A_ARG_TYPE_ID*) See section 2.3.4	""	
RunningEUIDs	*R*	**string**	CSV(*A_ARG_TYPE_ID*) See section 2.3.5	""	
ErrorEUIDs	*R*	**string**	CSV(*A_ARG_TYPE_ID*) See section 2.3.6	""	
A_ARG_TYPE_Boolean	*O*	**boolean**	(section 2.3.7)		
A_ARG_TYPE_String	*O*	**string**	(section 2.3.8)		
A_ARG_TYPE_ID	*R*	**ui4**	See section 2.3.9		
A_ARG_TYPE_IDs	*R*	**string**	CSV(**ui4**) See section 2.3.10		
A_ARG_TYPE_OperationState	*R*	**string**	*Requested*, *InProgress*, *Committed*, *Completed*, *Error* See section 2.3.11		
A_ARG_TYPE_URI	*R*	**uri**	See section 2.3.12		

Variable Name	Req. or Opt.[1]	Data Type	Allowed Value	Default Value [2]	Eng. Units
A_ARG_TYPE_Action	*R*	**string**	*Install*, *Update*, *Uninstall*, *Start*, *Stop* See section 2.3.13		
A_ARG_TYPE_ErrorDescription	*R*	**string**	*Error_None*, *Error_ConcurrentAccess*, *Error_MissingDependency*, *Error_Network*, *Error_CorruptedFile*, *Error_DiskFull*, *Error_Other* See section 2.3.14		
A_ARG_TYPE_DUType	*R*	**string**	*Firmware*, *Application*, *Configuration*, *Other* See section 2.3.15		
A_ARG_TYPE_Name	*O*	**string**	See section 2.3.16		
A_ARG_TYPE_Version	*O*	**string**	See section 2.3.17		
A_ARG_TYPE_DUState	*O*	**string**	*Installing*, *Unresolved*, *Installed*, *Uninstalling*, *Uninstalled* See section 2.3.18		
A_ARG_TYPE_EURequestedState	*O*	**string**	*Active*, *Inactive* See section 2.3.19		
A_ARG_TYPE_EURunningState	*O*	**string**	*Running*, *Stopped*, *Starting*, *Stopping* See section 2.3.20		
Non-standard state variables implemented by an UPnP vendor go here.	*X*	*TBD*	*TBD*	*TBD*	*TBD*

[1] R = Required, O = Optional, X = Non-standard.

2.3.1 *OperationIDs*

Comma-separated list of operation identifiers. It stores the *OperationID* list of *Requested* and *InProgress* operations (see section 2.6 for the definition of an operation).

This state variable is evented to notify control points when software management operations are requested or finished. When a software management operation is requested, its *OperationID* is added to the *OperationIDs* list. An operation is in the *Requested* state when an asynchronous action is successfully called. When the operation ends, its *OperationID* is removed from the *OperationIDs* list. An operation ends in the *Completed* or *Error* state when all the managed software entities reach a stable state (see section 2.6).

2.3.2 *DUIDs*

Comma-separated list of the DUIDs for all unresolved or installed DUs.

This state variable is evented to notify control points when new DUs enter the *Unresolved* or *Installed* or *Uninstalled* states. When an install operation is successfully completed (see section 2.6), every new *DUID* that appears is added to the *DUIDs* list. When an *Uninstall()* operation is successfully completed, every associated *DUID* is removed from the *DUIDs* list. The DUID 0 is reserved to identify the primary firmware, i.e., a firmware the physical device is booting on.

2.3.3 *EUIDs*

Comma-separated list of the EUIDs for all EUs.

This state variable is evented to notify control points when new EUs appear with the installation of DUs and when they disappear with the uninstallation of related DUs. When an *Install()* operation is successfully completed (see section 2.6), every new *EUID* that comes into existence is added to the *EUIDs* list. When an *Uninstall()* operation is successfully completed, every associated *EUID* is removed from the *EUIDs* list. The EUID 0 is reserved to identify the primary firmware, i.e., the firmware the physical device is booting on.

2.3.4 *ActiveEUIDs*

Comma-separated list of the EUIDs for all EUs that have explicitly been requested to start using the *Start()* action.

This state variable stores the *EUID* list of *Active* EUs. The elements in this list are contained in *EUIDs* list. It is evented to notify control points when EUs become *Active* and when they become *Inactive*. A *EUID* enters in the *ActiveEUIDs* list when the EU is successfully requested to start using the *Start()* action. A *EUID* leaves the *ActiveEUIDs* list when the EU is successfully requested to stop by the *Stop()* action.

2.3.5 *RunningEUIDs*

Comma-separated list of the EUIDs for all EUs that are currently running.

This state variable stores the *EUID* list of *Running* EUs. The elements in this list are contained in *EUIDs* list. It is evented to notify control points when EUs become *Running* and when they become *Stopped*. A *EUID* enters in the *RunningEUIDs* list when it is observed as running, i.e., is currently running whatever the means used to start it, i.e., by an explicit call of the *Start()* action or any other EE specific means. A *EUID* leaves the *RunningEUIDs* list when the EU is no more observed as running, i.e., is NOT currently running.

2.3.6 *ErrorEUIDs*

Comma-separated list of the EUIDs which are currently in an error state.

This state variable stores the EUID list of Error EUs. The elements in this list are contained in EUIDs list. It is evented to notify control points when EUs enter and leave *Error* state. The detailed error information MAY be retrieved by a control point from the *Software Data Model [*Appendix B Software Data Model (Normative)*]*.

2.3.7 *A_ARG_TYPE_Boolean*

A boolean argument.

2.3.8 *A_ARG_TYPE_String*

A string argument.

2.3.9 *A_ARG_TYPE_ID*

This state variable is introduced to provide type information for *OperationID*, *DUID* and *EUID* arguments in various actions. It is an unsigned integer with a minimum value of 0 that is assigned by the device.

2.3.10 *A_ARG_TYPE_IDs*

This state variable is introduced to provide type information for *TargetedIDs* argument in the *GetOperationInfo()* action.

2.3.11 *A_ARG_TYPE_OperationState*

This state variable is introduced to provide type information for the *OperationState* argument in the *GetOperationInfo()* action. Allowed values are: *Requested*, *InProgress*, *Completed*, *Error* (see Figure 2-4).

The *Requested* state is reached when an operation is created. The operation stays in this state until the *ParentDevice* is able to process it. One reason an operation could be delayed is the execution of a previously called operation.

Once the *ParentDevice* is able to process the operation the operation state is set to *InProgress*. It will stay in this state until the operation completes successfully or fails.

All the time an operation is in *Requested* or *InProgress* state, its *OperationID* is stored in the *OperationIDs* list (see 2.3.1).

Table 2-2: AllowedValueList for *A_ARG_TYPE_OperationState*

Value	Description	Req. or Opt.
Requested,		*R*
InProgress,		*R*
Completed,		*R*
Error		*R*

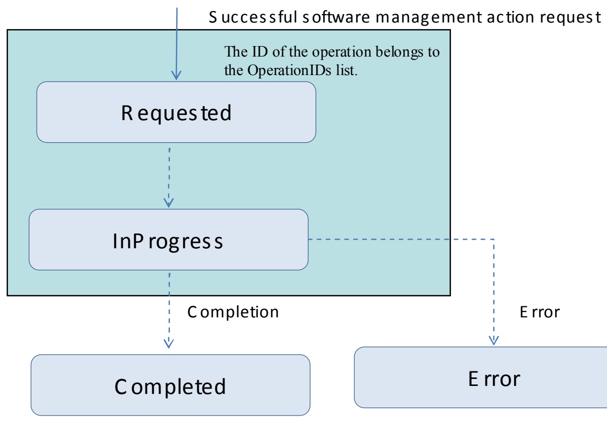

Successful software management action request
The ID of the operation belongs to the OperationIDs list.
Requested
InProgress
Completion
Error
Completed
Error

Figure 2-4 Software management operation state diagram

2.3.12 *A_ARG_TYPE_URI*

This state variable is introduced to provide type information for *URI* argument in various actions [UDA]. The value of a URI can also be an empty string (“”).

2.3.13 *A_ARG_TYPE_Action*

This state variable is introduced to provide type information for *Action* argument in the *GetOperationInfo()* action. Allowed values are names of defined software management actions: *Install*, *Update*, *Uninstall*, *Start*, *Stop* and the names of any vendor-specific asynchronous actions (see Section 2.2.5).

Table 2-3: AllowedValueList for *A_ARG_TYPE_Action*

Value	Req. or Opt.
Install	*R*
Update	*R*
Uninstall	*R*
Start	*R*
Stop	*R*
Vendor-defined	*X*[1]

[1]Vendor-specific allowed values must be listed in this table.

2.3.14 *A_ARG_TYPE_ErrorDescription*

This state variable is introduced to provide type information for the *ErrorDescription* argument in the *GetOperationInfo()* action. Allowed values are names describing the source of the failure of an operation:

- *Error_None*: no error to be described.
- *Error_ConcurrentAccess*: multiple concurrent accesses to the same resource.
- *Error_MissingDependency*: some dependency related error
- *Error_Network*: communication failed because of a network failure.
- *Error_CorruptedFile*: the file currently accessed is invalid for reading.
- *Error_StorageFull*: no space available on the persistent storage.
- *Error_Other*: all other specific errors.
- Vendor-specific values that have to be considered as an error case

Table 2-4: AllowedValueList for *A_ARG_TYPE_ErrorDescription*

Value	Req. or Opt.
Error_None	*R*
Error_ConcurrentAccess	*R*
Error_MissingDependency	*R*
Error_Network	*R*
Error_CorruptedFile	*R*
Error_DiskFull	*R*
Error Other	*R*
Vendor-defined	*X*[1]

[1]Vendor-specific allowed values must be listed in this table.

2.3.15 *A_ARG_TYPE_DUType*

This state variable is introduced to provide type information for *DUType* argument in various actions. SMS actions use input argument of type *A_ARG_TYPE_DUType* to indicate the type of the targeted DU.

Allowed types for a DU:

- *Firmware*: a DU which is a firmware
- *Application*: a DU which is an application for the targeted EE
- *Configuration*: a DU which contains configuration data
- *Other*: a DU which is of any other type
- Vendor-specific values

Table 2-5: AllowedValueList for *A_ARG_TYPE_DUType*

Value	Req. or Opt.
Firmware	*R*

Value	Req. or Opt.
Application	*R*
Configuration	*R*
Other	*R*
Vendor-defined	*X*[1]

[1]Vendor-specific allowed values must be listed in this table.

2.3.16 *A_ARG_TYPE_Name*

This state variable is introduced to provide type information for *DUName* and *EUName* arguments in various actions. Format of this value is EE specific.

2.3.17 *A_ARG_TYPE_Version*

This state variable is introduced to provide type information for *DUVersion* and *EUVersion* arguments in various actions. Format of this value is EE specific.

2.3.18 *A_ARG_TYPE_DUState*

This state variable is introduced to provide type information for *DUState* argument in the *GetDUInfo()* action.

Table 2-6: AllowedValueList for *A_ARG_TYPE_DUState*

Value	Req. or Opt.
Installing	*R*
Unresolved	*R*
Installed	*R*
Uninstalling	*R*
Uninstalled	*R*

2.3.19 *A_ARG_TYPE_EURequestedState*

This state variable is introduced to provide type information for *EURequestedState* argument in the *GetEUInfo()* action.

Table 2-7: AllowedValueList for *A_ARG_TYPE_EURequestedState*

Value	Req. or Opt.
Active	*R*
Inactive	*R*

2.3.20 A_ARG_TYPE_EURunningState

This state variable is introduced to provide type information for *EURunningState* argument in the *GetEUInfo()* action.

Table 2-8: AllowedValueList for A_ARG_TYPE_EURunningState

Value	Req. or Opt.
Running	*R*
Stopped	*R*
Starting	*R*
Stopping	*R*

2.4 Eventing and Moderation

Table 2-9: Event Moderation

Variable Name	Evented	Moderated Event	Max Event Rate[1]	Logical Combination	Min Delta per Event[2]
OperationIDs	*Yes*	*Yes*	*0.2*		
DUIDs	*Yes*	*Yes*	*1*		
EUIDs	*Yes*	*Yes*	*1*		
ActiveEUIDs	*Yes*	*Yes*	*1*		
RunningEUIDs	*Yes*	*Yes*	*0.2*		
ErrorEUIDs	*Yes*	*Yes*	*0.2*		
Non-standard state variables implemented by an UPnP vendor go here.	*TBD*	*TBD*	*TBD*	*TBD*	*TBD*

[1] Determined by N, where Rate = (Event)/(N secs).
[2] (N) * (allowedValueRange Step).

2.5 Actions

The following Sections contain detailed information about these actions, including short descriptions of the actions, the effects of the actions on state variables, and error codes defined by the actions.

Table 2-10: Actions

Name	Device R/O[1]	Control Point R/O
Install()	R	O
Update()	R	R
Uninstall()	R	O
Start()	R	O
Stop()	R	O
GetDUIDs()	R	O
GetEUIDs()	R	O
GetActiveEUIDs()	R	O
GetRunningEUIDs()	R	O
GetOperationInfo()	R	O
GetOperationIDs()	R	O
GetErrorEUIDs()	R	O
GetDUInfo()	O	O
GetEUInfo()	O	O
Non-standard actions implemented by an UPnP vendor go here.	X	X

[1] R = Required, O = Optional, X = Non-standard.

2.5.1 ***Install()***

The *Install()* action installs a DU. As a result of this action invocation a new *OperationID* is generated by the device and is returned. As it is an asynchronous action, details on this ongoing operation can be retrieved using the *GetOperationInfo()* action using the returned *OperationID*.

After successful installation, at least one DUID is generated and is returned to the control point using the DUIDs evented state variable. DUIDs MAY also be polled from the *Software Data Model* (see

Appendix B Software Data Model)

Depending on the targeted EE, the DU MAY reach the *Unresolved* state. This state corresponds to the successful installation of the DU with some of its dependencies unresolved. So after the successful completion of the install operation, the state of the DU will be either *Unresolved* or *Installed* (see Figure 2-4). Also on some EEs a DU MAY pass through a transitory state named *Installing* (see Figure 2-1).

2.5.1.1 Arguments

The input arguments are used as follows:

- **DUURI**: the URI of the DU that needs to be installed. The value MUST NOT be empty.
- **DUType**: the type of the DU provided by the control point as a hint to help in installation process. This is only a hint and need not match the actual DU type.
- **HandleDependencies**: boolean to trigger the dependency handling mechanism

The output arguments are defined as follows:

- **OperationID**: the identifier of the operation linked to the action call.

Table 2-11: Arguments for *Install()*

Argument	Direction	relatedStateVariable
DUURI	*IN*	*A_ARG_TYPE_URI*
DUType	*IN*	*A_ARG_TYPE_DUType*
HandleDependencies	*IN*	*A_ARG_TYPE_Boolean*
OperationID	*OUT*	*A_ARG_TYPE_ID*

2.5.1.2 Dependency on State

None.

2.5.1.3 Effect on State

When the *Install()* is called, a new *OperationID* is assigned to the Operation, which will be in the *Requested* state. The action returns with the *OperationID* output argument, and the OperationID is added to the *OperationIDs* list.

When the *ParentDevice* is able to process the operation, the *OperationState* will be assigned the *InProgress* state.

A DUID is created and a new instance with the *Installing* state MUST be added into the *Software Data Model*, if the data model is supported. If the *HandleDependencies* is "**0**", only the DU matching the *DUURI* MUST be targeted by the operation. If it is set to "**1**" and the device can handle dependencies, then all the dependent DUs also MUST be targeted by this operation. Each of the targeted DUs will be assigned a new *DUID*. As a side effect of the installation of a DU, a new DU instance sub-tree MAY be created for the DU in the *Software Data Model* with the DU state as *Installing*. The sub-tree MUST store information such as the URI of the DU. If any EU is associated to the DU, its corresponding sub-tree MUST also be created.

The operation is considered finished when all the targeted DUs are effectively installed or if any of installation fails. Only when the operation is successful, all the targeted DUs are assigned the *Installed* or *Unresolved* state. At this point the *DUID* of each *Installed* or *Unresolved* DU is added to the *DUIDs* list. Each related EU appears with the *Inactive* state. The IDs of these EUs are added to the *EUIDs* list.

Finally, the *OperationID* is removed from the *OperationIDs* list in the *Completed* state. Only one event message SHOULD be sent with the value of the three involved lists.

If the operation fails, all previously successfully installed DUs, i.e., DU in *Unresolved* or *Installed* state, remain in their state. The *OperationID* is removed from the *OperationIDs* list. The operation state is set to *Error* (see 2.3.14).

For some DUs, once the operation is *Completed* a reboot could be necessary. If the *BMS::SequenceMode* value is set to "**1**", the reboot SHOULD be avoided until *BMS::SequenceMode* value is set to "**0**".

2.5.1.4 Errors

Table 2-12: Error Codes for *Install()*

errorCode	errorDescription	Description
400-499	TBD	See UPnP Device Architecture section on Control.
500-599	TBD	See UPnP Device Architecture section on Control.
600-699	TBD	See UPnP Device Architecture section on Control.
701	Invalid URI	The *Install()* action failed because the *DUURI* argument is not valid, i.e., syntax base checking. For instance an empty value is not a valid DUURI for the *Install()* action.
702	HandleDependencies request not honored	The *Install()* action failed because *HandleDependencies* is set to "**1**" and the device is unable to handle the software entity dependencies or *HandleDependencies* is set to "**0**" and the device is handling software entity dependencies.
703	Already Installed, multiple Instances not allowed	The *Install()* action failed because an identical DU is already *Installed* or *Unresolved*.
704	Already in a transitory state	The action failed because the software entity is already in a transitory state.

2.5.2 ***Update()***

The *Update()* action updates a DU that was already installed and is either in *Unresolved* or *Installed* state. As a result of this action invocation a new *OperationID* is generated by the device and is returned. As it is an asynchronous action, details on the ongoing operation can be retrieved using the *GetOperationInfo()* action and the OperationID.

The action takes an optional URI input argument from which to update this particular DU. Here optional means that the value MAY be empty. If the NewDUURI is an empty string then the URI used during installation time or during the last successful update, will be used.

On successful update the DUID MUST NOT change and hence the old DUID is still maintained. The state of the DU may be either *Installed* or *Unresolved*.

2.5.2.1 Arguments

The input arguments are used as follows:

- **DUID**: the identifier of the DU that needs to be updated.
- **NewDUURI**: the URI of the DU that needs to be used to update the DU instead of the one used for the initial installation or the previous update. The value can be empty.
- **HandleDependencies**: boolean to trigger the dependency handling mechanism of the EE.

The output argument is defined as follows:

- **OperationID**: the identifier of the Operation linked to the action call.

Table 2-13: Arguments for *Update()*

Argument	Direction	relatedStateVariable
DUID	*IN*	*A_ARG_TYPE_ID*
NewDUURI	*IN*	*A_ARG_TYPE_URI*
HandleDependencies	*IN*	*A_ARG_TYPE_Boolean*
OperationID	*OUT*	*A_ARG_TYPE_ID*

2.5.2.2 Dependency on State

The DU with given *DUID* MUST exist and be assigned the *Installed* or the *Unresolved* state.

2.5.2.3 Effect on State

A new *OperationID* is assigned to the Operation, which will be in the *Requested* state. The action returns with the *OperationID* output argument and the OperationID is also added to the *OperationIDs* list.

If *HandleDependencies* is set to "**0**", only the DU matching the *DUID* is targeted by the operation. If it is set to "**1**", and the device can handle dependencies then, all the dependent DUs also must be targeted by this operation. The details of how a device decides which dependent DUs to be targeted for updating is implementation dependent.

A new URI MAY be given as the *NewDUURI* input argument. In that case, the DU is updated from this location. If the argument is set with an empty string, the device updates the DU from a location that is already known by the device, e.g., the URI argument provided to *Install()* action.

The operation is considered *Completed* when:

- all the related EUs which were running are effectively stopped
- all the targeted DUs are effectively updated, i.e., updated, installed or uninstalled
- all the related EUs which were *Active* have been requested to start.

The operation is considered in *Error* state when:

- any of the conditions to reach the *Completed* state is not satisfied.

Final stable state reached by the *ParentDevice* MUST reflect the following:

- all the targeted DUs are assigned the appropriate *Installed* or *Uninstalled* state,
- new DUIDs, if any, are added to the *DUIDs* list,

- new EUs, if any, appear in the *Inactive* state (*A control point expects all the EUs to be in their same state as before the update, but in certain circumstances because of for eg: security issues or because of EE behavior the previously running EUs may not be running*),
- the IDs of the new EUs are added to the *EUIDs* list,
- the IDs of the EUs that were successfully started are added to the *RunningEUIDs* list,
- uninstalled DUIDs, if any, are removed from the *DUIDs* list,
- EUIDs related to any uninstalled DUs are removed from the *EUIDs*, the *ActiveEUIDs* and the *RunningEUIDs* lists,
- the value of the URI *Parameter* available in the Software Data Model MUST be updated with the value of the NewDUURI argument if it was specified, i.e., not empty and if the software data model is supported.
- the *OperationID* is removed from the *OperationIDs* list in the *Completed* or *Error* state.
- only one event message SHOULD be sent with the value of the five involved lists.

If *HandleDependencies* is set to "**0**" or if no new DU is installed or removed, the DUIDs list is not modified which means no install or uninstall of DUs.

For some DUs, once the operation is *Completed* a reboot could be needed. If the *BMS::SequenceMode* value is set to "**1**", the reboot SHOULD be avoided until *BMS::SequenceMode* value is set to "**0**".

As a side effect of this action the SMS data model MAY need to be updated. If so, this update MUST be done before any of the related state variables are updated.

2.5.2.4 Errors

Table 2-14: Error Codes for *Update()*

errorCode	errorDescription	Description
400-499	TBD	See UPnP Device Architecture section on Control.
500-599	TBD	See UPnP Device Architecture section on Control.
600-699	TBD	See UPnP Device Architecture section on Control.
701	Invalid URI	The *Update()* action failed because the *NewDUURI* argument is not valid i.e., syntax base checking. For instance the empty value is not a valid DUURI for the *Update()* action if no alternative valid URI is known by the device.
702	HandleDependencies request not honored	The *Update()* action failed because *HandleDependencies* is set to "**1**" and the device is unable to handle the software entity dependencies or *HandleDependencies* is set to "**0**" and the device is handling software entity dependencies.
704	Already in a transitory state	The action failed because the software entity is already in a transitory state.
705	Invalid DUID	The *Update()* action failed because the *DUID* argument is not valid.
710	NewDUURI not supported	The update() action failed because the NewDUURI is not supported by the EE.

2.5.3 Uninstall()

The *Uninstall()* action uninstalls any DU that is already installed. As a result of this action invocation a new *OperationID* is generated by the device and is returned. As it is an asynchronous action, details on the ongoing operation can be retrieved using the *GetOperationInfo()* action and the OperationID.

Upon successful uninstall the state of the DU becomes *Uninstalled*. This state of the DU MAY be maintained in the device for an implementation-specific amount of time to enable a control point to read this state. After a while, the DUID MAY be removed and become invalid.

2.5.3.1 Arguments

The input arguments are used as follows:

- **DUID**: the identifier of the DU that needs to be uninstalled.
- **HandleDependencies**: boolean to trigger the dependency handling mechanism of the EE.

The output argument is defined as follows:

- **OperationID**: the identifier of the operation linked to the action call.

Table 2-15: Arguments for *Uninstall()*

Argument	Direction	relatedStateVariable
DUID	*IN*	*A_ARG_TYPE_ID*
HandleDependencies	*IN*	*A_ARG_TYPE_Boolean*
OperationID	*OUT*	*A_ARG_TYPE_ID*

2.5.3.2 Dependency on State

The given *DUID* MUST exist and identify a DU in the *Installed* or the *Unresolved* state.

2.5.3.3 Effect on State

A new *OperationID* is assigned to the Operation, which will be in the *Requested* state. The action returns with the *OperationID* output argument and the OperationID is also added to the *OperationIDs* list.

If the *HandleDependencies* is set to "**0**", only the DU matching the *DUID* is targeted by the operation. If it is set to "**1**", and the device can handle dependencies then, all the dependent DUs also must be targeted by this operation. The details of how a device decides which dependent DUs to be targeted for uninstalling is implementation dependent. Each of the targeted DUs is assigned the *Uninstalling* state. *Running* EUs which are related to the targeted DUs are assigned the *Stopping* state.

The operation is considered *Completed* when:

- all the targeted EUs which were *Running* are stopped,
- all the targeted DUs are effectively uninstalled

The operation is considered *Error* when:

- one of the conditions to reach the *Completed* state is not satisfied.

The *DUID* of each *Uninstalled* DU is removed from the *DUIDs* list. Related EUs are removed from the *EUIDs*, *ActiveEUIDs*, and *RunningEUIDs* lists. Finally, the *OperationID* is removed from the *OperationIDs* list in the *Completed* or *Error* state. Only one event message SHOULD be sent with the value of all involved lists.

For some DUs, once the operation is *Completed* a reboot could be needed. If the *BMS::SequenceMode* value is set to "**1**", the reboot SHOULD be avoided until *BMS::SequenceMode* value is set to **0**.

As a side effect of this action the SMS data model MAY be updated. If so, this update MUST be done before any of the above mentioned state variables are updated.

2.5.3.4 Errors

Table 2-16: Error Codes for *Uninstall()*

errorCode	errorDescription	Description
400-499	TBD	See UPnP Device Architecture section on Control.
500-599	TBD	See UPnP Device Architecture section on Control.
600-699	TBD	See UPnP Device Architecture section on Control.
702	HandleDependencies request not honored	The *Uninstall()* action failed because *HandleDependencies* is set to "**1**" and the device is unable to handle the software entity dependencies or *HandleDependencies* is set to "**0**" and the device is handling software entity dependencies.
704	Already in a transitory state	The action failed because the software entity is already in a transitory state.
705	Invalid DUID	The *Uninstall()* action failed because the *DUID* argument is not valid.

2.5.4 ***Start()***

The *Start()* action starts an EU. As it is an asynchronous action, details on the ongoing operation can be retrieved using the *GetOperationInfo()* action and the *OperationID*.

When a DU is installed a set of EUs will appear in *Inactive* state. By invoking a *Start()* action on these EUs, a control point makes them *Active*, i.e., asks the EUs to be started. Hence started EUs will be observed as *Running.*

2.5.4.1 Arguments

The input arguments are used as follows:

- **EUID**: the identifier of the EU to be started.
- **HandleDependencies**: boolean to trigger the dependency handling mechanism of the EE.

The output argument is defined as follows:

- **OperationID**: the identifier of the operation linked to the action call.

Table 2-17: Arguments for *Start()*

Argument	Direction	relatedStateVariable
EUID	*IN*	*A_ARG_TYPE_ID*
HandleDependencies	*IN*	*A_ARG_TYPE_Boolean*
OperationID	*OUT*	*A_ARG_TYPE_ID*

2.5.4.2 Dependency on State

The given *EUID* MUST exist and be assigned the *Inactive* state.

2.5.4.3 Effect on State

A new *OperationID* is assigned the *Requested* value and enters the *OperationIDs* list. The action returns with the *OperationID* output argument.

When *Software Data Model* is implemented, this action sets the `/UPnP/DM/Software/DU/#/EU/#/RequestedState` to reflect that an EU is *Active*.

If the *HandleDependencies* is set to "**0**", only the EU matching the *EUID* is targeted by the operation. If it is set to "**1**", and the device can handle dependencies then, all the dependent EUs also must be targeted by this operation. The details of how a device decides which dependent EUs to be targeted for starting is implementation dependent. Each of the targeted EUs is assigned the *Active* state.

The operation is considered finished when all the targeted EUs are *Active* and were requested to start. If the operation is successful, all the targeted EUs are observed in the *Running* state.

The *EUID* of each *Active* EU is added to the *ActiveEUIDs* list. The *EUID* of each EU running is added to the *RunningEUIDs* list.

If the operation is successful, the *OperationID* is removed from the *OperationIDs* list in the *Completed* state.

If the operation fails the *OperationID* removed from the *OperationIDs* list in the *Error* state.

Only one event message SHOULD be sent with the value of all involved lists.

For some EUs, once the operation is *Completed* a reboot could be needed. If the *BMS::SequenceMode* value is set to **1**, the reboot SHOULD be avoided until *BMS::SequenceMode* value is set to **0**.

As a side effect of this action the SMS data model MAY be updated. If so, this update MUST be done before any of the above mentioned state variables are updated.

2.5.4.4 Errors

Table 2-18: Error Codes for *Start()*

errorCode	errorDescription	Description
400-499	TBD	See UPnP Device Architecture section on Control.
500-599	TBD	See UPnP Device Architecture section on Control.
600-699	TBD	See UPnP Device Architecture section on Control.

errorCode	errorDescription	Description
702	HandleDependencies request not honored	The *Start()* action failed because *HandleDependencies* is set to "**1**" and the device is unable to handle the software entity dependencies or *HandleDependencies* is set to "**0**" and the device is handling software entity dependencies.
704	Already in a transitory state	The action failed because the software entity is already in a transitory state.
706	Invalid EUID	The *Start()* action failed because the *EUID* argument is not valid.
709	DU is unresolved	The *Start()* action failed because the DU related to this EU is still *Unresolved*.

2.5.5 ***Stop()***

The *Stop()* action stops an EU that was already running. As it is an asynchronous action, details on the ongoing operation can be retrieved using the *GetOperationInfo()* action and the OperationID.

By invoking the stop action, the state of an *Active* EU will be changed to *Inactive*, i.e., asked to be stopped. Hence the EUID of all the EUs successfully *Stopped* will leave the *RunningEUIDs* list.

2.5.5.1 Arguments

The input arguments are used as follows:

- **EUID**: the identifier of the EU to be stopped.
- **HandleDependencies**: boolean to trigger the dependency handling mechanism of the EE.

The output argument is defined as follows:

- **OperationID**: the identifier of the operation linked to the action call.

Table 2-19: Arguments for *Stop()*

Argument	Direction	relatedStateVariable
EUID	*IN*	*A_ARG_TYPE_ID*
HandleDependencies	*IN*	*A_ARG_TYPE_Boolean*
OperationID	*OUT*	*A_ARG_TYPE_ID*

2.5.5.2 Dependency on State

The given *EUID* MUST exist and be assigned the *Active* state.

2.5.5.3 Effect on State

A new *OperationID* is assigned the *Requested* value and enters the *OperationIDs* list. The action returns with the *OperationID* output argument.

When *Software Data Model* is implemented, this action sets the `/UPnP/DM/Software/DU/#/EU/#/RequestedState` to reflect that an EU is *Inactive*.

If the *HandleDependencies* is set to "**0**", only the EU matching the *EUID* is targeted by the operation. If it is set to "**1**", and the device can handle dependencies then, all the dependent EUs also must be targeted by this operation. The details of how a device decides which dependent EUs to be targeted for stopping is implementation dependent. Each of the targeted EUs is assigned the *Inactive* state.

The action is considered finished when all the targeted EUs are *Inactive* and were requested to stop.

If the operation is successful, all the targeted EUs are assigned the *Inactive* state and cease to be running. The *EUID* of each *Inactive* EU is removed from the *ActiveEUIDs* list. The EUID of each Stopped EU is removed from the *RunningEUIDs* list. Finally, the *OperationID* is removed from the *OperationIDs* list in the *Completed* or *Error* state. Only one event message SHOULD be sent with the value of all involved lists.

For some EUs once the operation is *Completed* a reboot could be needed. If the *BMS::SequenceMode* value is set to **1**, the reboot SHOULD be avoided until *BMS::SequenceMode* value is set to **0**.

As a side effect of this action the SMS data model MAY be updated. If so, this update MUST be done before any of the related state variables are updated.

2.5.5.4 Errors

Table 2-20: Error Codes for *Stop ()*

errorCode	errorDescription	Description
400-499	TBD	See UPnP Device Architecture section on Control.
500-599	TBD	See UPnP Device Architecture section on Control.
600-699	TBD	See UPnP Device Architecture section on Control.
702	HandleDependencies request not honored	The *Stop()* action failed because *HandleDependencies* is set to "**1**" and the device is unable to handle the software entity dependencies or *HandleDependencies* is set to "**0**" and the device is handling software entity dependencies.
704	Already in a transitory state	The action failed because the software entity is already in a transitory state.
706	Invalid EUID	The *Stop()* action failed because the *EUID* argument is not valid.

2.5.6 ***GetDUIDs()***

The *GetDUIDs()* action return the current value of the *DUIDs* state variable.

2.5.6.1 Arguments

The output argument is defined as follows:

- **DUIDs**: the identifiers of the installed DUs. If no DU is in *Installed* or *Unresolved* state the returned value is an empty string.

Table 2-21: Arguments for *GetDUIDs()*

Argument	Direction	relatedStateVariable
DUIDs	*OUT*	*DUIDs*

2.5.6.2 Dependency on State

None.

2.5.6.3 Effect on State

None.

2.5.6.4 Errors

Table 2-22: Error Codes for *GetDUIDs()*

errorCode	errorDescription	Description
400-499	TBD	See UPnP Device Architecture section on Control.
500-599	TBD	See UPnP Device Architecture section on Control.
600-699	TBD	See UPnP Device Architecture section on Control.

2.5.7 ***GetEUIDs()***

The *GetEUIDs()* action returns the current value of the *EUIDs* state variable.

2.5.7.1 Arguments

The output argument is defined as follows:

- **EUIDs**: the identifiers of all the installed EUs.

Table 2-23: Arguments for *GetEUIDs()*

Argument	Direction	relatedStateVariable
EUIDs	*OUT*	*EUIDs*

2.5.7.2 Dependency on State

None.

2.5.7.3 Effect on State

None.

2.5.7.4 Errors

Table 2-24: Error Codes for *GetEUIDs()*

errorCode	errorDescription	Description
400-499	TBD	See UPnP Device Architecture section on Control.
500-599	TBD	See UPnP Device Architecture section on Control.
600-699	TBD	See UPnP Device Architecture section on Control.

2.5.8 GetActiveEUIDs()

The *GetActiveEUIDs()* action returns the current value of the *ActiveEUIDs* state variable.

2.5.8.1 Arguments

The output argument is defined as follows:

- **ActiveEUIDs**: the identifiers of all the EUs that are in *Active* state.

Table 2-25: Arguments for GetActiveEUIDs()

Argument	Direction	relatedStateVariable
ActiveEUIDs	*OUT*	*ActiveEUIDs*

2.5.8.2 Dependency on State

None.

2.5.8.3 Effect on State

None.

2.5.8.4 Errors

Table 2-26: Error Codes for GetActiveEUIDs()

errorCode	errorDescription	Description
400-499	TBD	See UPnP Device Architecture section on Control.
500-599	TBD	See UPnP Device Architecture section on Control.
600-699	TBD	See UPnP Device Architecture section on Control.

2.5.9 GetRunningEUIDs()

The *GetRunningEUIDs()* action returns the current value of the *RunningEUIDs* state variable.

2.5.9.1 Arguments

The output argument is defined as follows:

- **RunningEUIDs**: the identifiers of all the EUs that are in *Running* state.

Table 2-27: Arguments for GetRunningEUIDs()

Argument	Direction	relatedStateVariable
RunningEUIDs	*OUT*	*RunningEUIDs*

2.5.9.2 Dependency on State

None.

2.5.9.3 Effect on State

None.

2.5.9.4 Errors

Table 2-28: Error Codes for *GetRunningEUIDs()*

errorCode	errorDescription	Description
400-499	TBD	See UPnP Device Architecture section on Control.
500-599	TBD	See UPnP Device Architecture section on Control.
600-699	TBD	See UPnP Device Architecture section on Control.

2.5.10 *GetOperationInfo()*

The *GetOperationInfo()* action gets the information about an operation identified by an *OperationID*.

Once an operation has been successfully requested, the operation info SHOULD remain available at least until a disappearance / reappearance of the *SoftwareManagement:1* service on the network, e.g., as a result of a power cycle. An implementation MAY drop the oldest *OperationIDs* from the list because of memory constraint. An implementation MAY retain operation info across a service restart.

2.5.10.1 Arguments

The input argument is used as follows:

- **OperationID**: the identifier of the targeted operation.

The output arguments are defines as follows:

- **OperationState**: the state of the operation.
- **TargetedIDs**: the IDs of the DUs or the EUs targeted by the operation. The first ID MUST identify the software entity explicitly targeted by the operation, e.g., the EUID passed in to the *Start()* action. If the operation state is *Requested*, *InProgress* or *Error* the *TargetedIDs* list MAY be incomplete and possibly empty. When the operation is *Completed*, the list MUST be exhaustive.
- **Action:** the action that has initiated the operation.
- **ErrorDescription**: detailed description of error state. In case of completion without error, *Error_None* MUST be returned.
- **AdditionalInfo**: an informative description of the result of the operation.

Table 2-29: Arguments for *GetOperationInfo()*

Argument	Direction	relatedStateVariable
OperationID	*IN*	*A_ARG_TYPE_ID*
OperationState	*OUT*	*A_ARG_TYPE_OperationState*
TargetedIDs	*OUT*	*A_ARG_TYPE_IDs*
Action	*OUT*	*A_ARG_TYPE_Action*
ErrorDescription	*OUT*	*A_ARG_TYPE_ErrorDescription*
AdditionalInfo	*OUT*	*A_ARG_TYPE_String*

2.5.10.2 Dependency on State

The given *OperationID* MUST exist.

2.5.10.3 Effect on State

None.

2.5.10.4 Errors

Table 2-30: Error Codes for *GetOperationInfo()*

errorCode	errorDescription	Description
400-499	TBD	See UPnP Device Architecture section on Control.
500-599	TBD	See UPnP Device Architecture section on Control.
600-699	TBD	See UPnP Device Architecture section on Control.
708	Invalid Operation ID	The *GetOperationInfo()* action failed because the *OperationID* argument is not valid.

2.5.11 ***GetOperationIDs()***

The *GetOperationIDs()* action returns the current value of the *OperationIDs* state variable.

2.5.11.1 Arguments

The output argument is defined as follows:

- **OperationIDs**: the identifiers of all the on going Operations i.e., Operations in either *Requested* or *InProgress* state.

Table 2-31: Arguments for *GetOperationIDs()*

Argument	Direction	relatedStateVariable
OperationIDs	*OUT*	*OperationIDs*

2.5.11.2 Dependency on State

None.

2.5.11.3 *Effect on State*

None.

2.5.11.4 *Errors*

Table 2-32: Error Codes for *GetOperationIDs()*

errorCode	errorDescription	Description
400-499	TBD	See UPnP Device Architecture section on Control.
500-599	TBD	See UPnP Device Architecture section on Control.
600-699	TBD	See UPnP Device Architecture section on Control.

2.5.12 *GetErrorEUIDs()*

The *GetErrorEUIDs()* action returns the current value of the *ErrorEUIDs* state variable.

2.5.12.1 *Arguments*

The output argument is defined as follows:

- *ErrorEUIDs*: a comma-separated list of EUIDs in an error state. This MAY be empty if and only if there are no EUs currently in error state.

Table 2-33: Arguments for *GetErrorEUIDs()*

Argument	Direction	relatedStateVariable
ErrorEUIDs	*OUT*	*ErrorEUIDs*

2.5.12.2 *Dependency on State*

None.

2.5.12.3 *Effect on State*

None.

2.5.12.4 *Errors*

Table 2-34: Error Codes for *GetErrorEUIDs()*

errorCode	errorDescription	Description
400-499	TBD	See UPnP Device Architecture section on Control.
500-599	TBD	See UPnP Device Architecture section on Control.
600-699	TBD	See UPnP Device Architecture section on Control.

2.5.13 *GetDUInfo()*

The *GetDUInfo()* action gets the information about a DU identified by its DUID.

Information about a DU MUST be available as soon as the DU is Installed. Information about the DU is removed as soon as the DU is Uninstalled.

2.5.13.1 Arguments

The input argument is used as follows:

- **DUID**: the identifier of the DU.

The output arguments are defined as follows:

- **DUName:** the name of the DU.
- **DUVersion:** the version of the DU.
- **DUType:** the type of the DU.
- **DUState:** the state of the DU.
- **DUURI:** the URI of the DU from where it is installed or updated. The value MUST NOT be empty.

Table 2-35: Arguments for GetDUInfo()

Argument	Direction	relatedStateVariable
DUID	*IN*	*A_ARG_TYPE_ID*
DUName	*OUT*	*A_ARG_TYPE_Name*
DUVersion	*OUT*	*A_ARG_TYPE_Version*
DUType	*OUT*	*A_ARG_TYPE_DUType*
DUState	*OUT*	*A_ARG_TYPE_DUState*
DUURI	*OUT*	*A_ARG_TYPE_URI*

2.5.13.2 Dependency on State

The given *DUID* MUST exist.

2.5.13.3 Effect on State

None.

2.5.13.4 Errors

Table 2-36: Error Codes for GetDUInfo()

errorCode	errorDescription	Description
400-499	TBD	See UPnP Device Architecture section on Control.
500-599	TBD	See UPnP Device Architecture section on Control.
600-699	TBD	See UPnP Device Architecture section on Control.
705	Invalid DUID	The *GetDUInfo()* action failed because the *DUID* argument is not valid.

2.5.14 GetEUInfo()

The *GetEUInfo()* action gets the information about an EU identified by its EUID.

Information about an EU MUST be available as soon as the related DU is installed successfully because EUs appear on successful installation of DUs. Information about the EU will be removed as soon as the corresponding DU is uninstalled.

2.5.14.1 Arguments

The input argument is used as follows:

- **EUID**: the identifier of the EU.

The output arguments are defined as follows:

- **EUName:** the name of the EU.
- **EUVersion:** the version of the EU.
- **EURequestedState:** the requested state of the EU.
- **EURunningState:** the running state of the EU.

Table 2-37: Arguments for GetEUInfo()

Argument	Direction	relatedStateVariable
EUID	*IN*	*A_ARG_TYPE_ID*
EUName	*OUT*	*A_ARG_TYPE_Name*
EUVersion	*OUT*	*A_ARG_TYPE_Version*
EURequestedState	*OUT*	*A_ARG_TYPE_EURequestedState*
EURunningState	*OUT*	*A_ARG_TYPE_EURunningState*

2.5.14.2 Dependency on State

The given *EUID* MUST exist.

2.5.14.3 Effect on State

None.

2.5.14.4 Errors

Table 2-38: Error Codes for GetEUInfo()

errorCode	errorDescription	Description
400-499	TBD	See UPnP Device Architecture section on Control.
500-599	TBD	See UPnP Device Architecture section on Control.
600-699	TBD	See UPnP Device Architecture section on Control.
706	Invalid EUID	The *GetEUInfo()* action failed because the *EUID* argument is not valid.

2.5.15 Common Error Codes

The following table lists error codes common to actions for this service type. If an action results in multiple errors, the most specific error must be returned.

Table 2-39: Common Error Codes

errorCode	errorDescription	Description
400-499	TBD	See UPnP Device Architecture section on Control.
500-599	TBD	See UPnP Device Architecture section on Control.
600-699	TBD	Common action errors. Defined by UPnP Forum Technical Committee.
700		Reserved for future extensions.
701	Invalid URI	The action failed because the given URI argument is not valid, i.e., syntax base checking
702	HandleDependencies request not honored	The action failed because *HandleDependencies* is set to "**1**" and the device is unable to handle the software entity dependencies or *HandleDependencies* is set to "**0**" and the device is handling software entity dependencies.
703	Already Installed, multiple Instances not allowed	The *Install()* action failed because an identical DU is already *Installed* or *Unresolved*.
704	Already in a transitory state	The action failed because the software entity is already in a transitory state.
705	Invalid DUID	The action failed because the given *DUID* value is not valid.
706	Invalid EUID	The action failed because the given *EUID* value is not valid.
707	Already in the targeted state	The action failed because the software entity is already in the targeted state.
708	Invalid Operation ID	The action failed because the given *OperationID* value is not valid.
709	DU is unresolved	The *Start()* action failed because the DU related to this EU is still *Unresolved*.
710	*NewDUURI* not supported	The *Update()* action failed because the *NewDUURI* is not supported by the EE.
800-899	*TBD*	*(Specified by UPnP vendor.)*

2.6 Theory of Operation

This informative section explains various scenarios to illustrate the actions that are supported by *SoftwareManagement:1* Service. The *SoftwareManagement:1* Service will support the following actions: *Install()*, *Update()*, *Uninstall()*, *Start()* and *Stop()* on software entities.

2.6.1 Scenarios

2.6.1.1 Dependency Handling

Dependency handling means one software entity being dependent on another software entity. For example a DU may be dependent on other DUs and an EU may be dependent on other EUs. Dependency handling is a capability of the EE. For example, an OSGi EE is able to handle dependencies. The properties `/UPnP/DM/Software/Capabilities/AbleToHandleDUDependency` and `/UPnP/DM/Software/Capabilities/AbleToHandleEUDependency` are used to indicate the capabilities. How SMS handles the dependencies is implementation specific. Below are some possible outcomes (Note: first DU is the DU on which the action is invoked):

1) During **installation** of a first DU, dependency handling may install other DUs as well. If dependent DUs are not installed, the first DU will be in *Unresolved* state else it will be in *Installed* state.

2) During **update** of a first DU, dependency handling may uninstall some DUs which are not in use currently and were installed because of dependency during installation and install some more new DUs. The running EUs corresponding to the updated DUs are stopped and restarted.

3) During **uninstall** of a first DU, dependency handling may uninstall some DUs which are not in use currently and were installed because of dependency during installation of first DU. EUs corresponding to the uninstalled DUs are stopped.

4) During **start** of a first EU, dependency handling may start some EUs on which the first EU is dependent upon.

5) During **stop** of a first EU, dependency handling may stop some EUs which are not in use currently and were started because of dependency handling during start of first EU.

2.6.1.2 Installing a Software entity successfully

1. Without dependency

A control point invokes the install action giving the URI of a first DU to be installed on to an EE. A URI is most probably a URL for practical purposes. After receiving the install action, the *ParentDevice* will create an operation and return the *OperationID*. This *OperationID* is added to the *OperationIDs* state variable and is evented. Once the operation is successfully executed, the first DU appears in *Installed* state (assuming no dependency). A new *DUID* is assigned to this DU, data model MAY be updated and this is added to the *DUIDs* state variable. All the EUs associated to this first DU appear in the *Inactive* state and their IDs are added to *EUIDs* state variable. The operation is successful and the *OperationID* is removed from the *OperationIDs* list. All the state variables are evented in a single event.

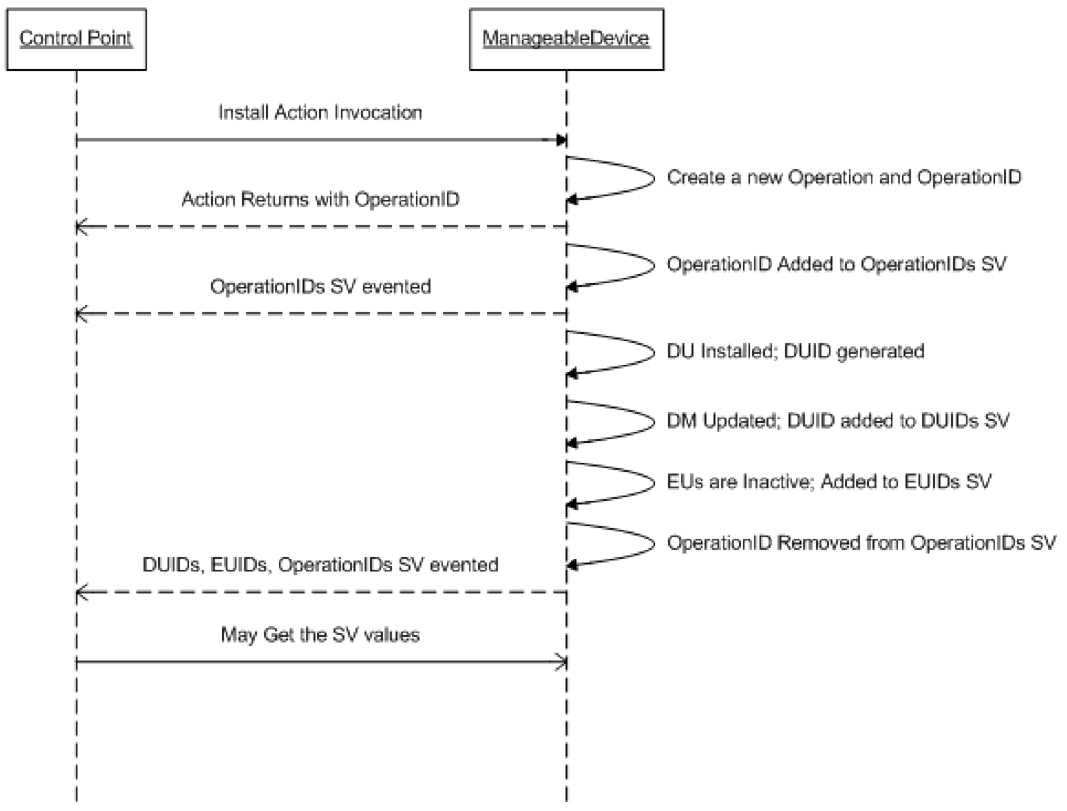

Control Point
ManageableDevice
Install Action Invocation
Create a new Operation and OperationID
Action Returns with OperationID
OperationID Added to OperationIDs SV
OperationIDs SV evented
DU Installed; DUID generated
DM Updated; DUID added to DUIDs SV
EUs are Inactive; Added to EUIDs SV
OperationID Removed from OperationIDs SV
DUIDs, EUIDs, OperationIDs SV evented
May Get the SV values

Figure 2-5 Installing a software entity successfully without dependency

2. **With Dependency**

A control point invokes the install action giving the URI of a first DU to be installed on to an EE along with dependency handling. With dependency handling on successful completion of an install operation all dependent DUs will be either installed or unresolved in addition to the first DU which will also be either in the installed or unresolved state. All EUs corresponding to the DUs will appear in the *Inactive* state. Other flow remains the same as above.

2.6.1.3 Installing a software entity (failure case)

A control point invokes the install action giving the URI of a first DU to be installed on to an EE along with dependency handling. If an *Error* occurs because of a dependent DU, the installation proceeds with other DUs without effecting the operation. But if an *Error* occurs because of the first DU and some of its dependent DUs are already installed, in that case implementations may choose to rollback and hence uninstall the dependent DUs installed in this operation. Note that the order of installation of DUs is implementation dependent. In any case control points depend on the evented state variables *DUIDs*, *EUIDs*, *ActiveEUIDs* and *OperationIDs*. Even in the failure case these state variables will be updated and control points will be in sync with the *ParentDevice*.

2.6.1.4 Update a Software entity successfully

A software entity can be a piece of software or firmware represented as a DU. For update a firmware is treated as a special kind of software. A firmware can be envisioned in terms of DU and EU. The primary firmware is identified using the ID "0" for both *DUID* and *EUID*. So updating a primary firmware is done by calling the *Update()* action on *DUID* "0".

Treating firmware as a common software entity gives the flexibility of designing a monolithic firmware as well as a modular firmware. An implementation is free to choose a set of DUs and EUs as firmwares. In any case DUID and EUID "0" are reserved for the primary firmware, i.e., the firmware the physical device is booting on.

Below are two scenarios with dependency and without dependency for updating a software entity.

1. **Without Dependency**

   A control point invokes the *Update()* action by optionally giving the new URI from where to update an already installed/ unresolved DU on to an EE. After receiving the *Update()*action, the SMS will create an operation and return the *OperationID*. This *OperationID* is added to the *OperationIDs* state variable and is evented. The already installed or unresolved DU will be uninstalled and a new DU will be installed from the new location or from a previously stored location. The previous *DUID* is retained. After a successful update the DU can be either unresolved or installed. Any related EUs which are running will be stopped. After update of a DU all the related EUs will appear in *Inactive* state irrespective of their previous state. These EUs will be treated as new EUs and are having new *EUID*s. The *OperationID* is removed from the *OperationIDs* list. The data model MAY be updated and the related state variables are updated for eventing.

2. **With Dependency**

   A control point invokes the *Update()* action optionally giving the new URI from where to update an already installed/unresolved DU on to an EE along with dependency handling. After receiving the *Update()* action, the SMS will create an operation and return the *OperationID*. This *OperationID* is added to the *OperationIDs* state variable and is evented. The already installed or unresolved DU will be uninstalled and a new DU will be installed from the new location or from a previously stored location. The previous *DUID* is retained. The same is repeated for all dependent DUs. With dependency handling, on successful completion of an update operation, all dependent DUs will be either installed or unresolved in addition to the first DU which will also be either in the installed or unresolved state. All the related EUs which are running will be stopped. After update all the related EUs will appear in *Inactive* state irrespective of their previous state. These EUs will be treated as new EUs and are having new *EUIDs*. The *OperationID* is removed from the *OperationIDs* list. The data model MAY be updated and the related state variables are updated for eventing.

2.6.1.5 Updating a software entity (failure case)

A control point invokes the *Update()* action by optionally giving the new URI from where to update an already installed/unresolved DU on to an EE along with dependency handling. If during update an ***Error*** occurs because of a dependent DU, the update proceeds with other DUs without effecting the operation. But if an ***Error*** occurs because of the first DU and some of its dependent DUs are already updated, in that case implementations may choose to rollback and hence uninstall those dependent DUs installed in this operation. Note that the order of installation of DUs is implementation dependent. In any case control points depend on the evented state variables *DUIDs*, *EUIDs*, *ActiveEUIDs* and *OperationIDs*. Even in the failure case these state variables will be updated and control points will be in sync with the *ParentDevice*.

2.6.1.6 Uninstall a Software entity successfully

1. Without Dependency

A control point invokes the *Uninstall()* action giving the DUID of a DU to be uninstalled from an EE. After receiving the *Uninstall()* action, the *ParentDevice* will create an operation and return the *OperationID*. This *OperationID* is added to the *OperationIDs* state variable and is evented. The operation is successfully completed once the DU corresponding to the *DUID* is uninstalled and that DU is removed from the *DUIDs* list. The EUs corresponding to the DU are stopped and removed from the *EUIDs* and *RunningEUIDs* list. The data model MAY be updated and *OperationID* is removed from *OperationIDs* list. All the state variables are evented.

2. With Dependency

A control point invokes the *Uninstall()* action giving the DUID of a first DU to be uninstalled from an EE along with dependency handling. After receiving the *Uninstall()* action, the *ParentDevice* will create an operation and return the *OperationID*. This *OperationID* is added to the *OperationIDs* state variable and is evented. The operation is successfully completed once all the DUs which are installed because of dependency handling will be uninstalled along with the first DU. A dependent DU will be uninstalled if and only if no more dependencies exist on it. All DUs are removed from the *DUIDs* list. All the EUs corresponding to the DUs are stopped and removed from the *EUIDs* and *RunningEUIDs* list. The *OperationID* is removed from *OperationIDs* list. The data model MAY be updated and state variables are evented.

2.6.1.7 Uninstall a software entity (failure case)

A control point invokes the *Uninstall()* action giving the *DUID* of a first DU to be uninstalled from an EE along with dependency handling. After receiving the uninstall action, the *ParentDevice* will create an operation and return the *OperationID*. This *OperationID* is added to the *OperationIDs* state variable and is evented. If during uninstall an ***Error*** occurs because of a dependent DU, the uninstall proceeds with other DUs without effecting the operation. But if an ***Error*** occurs because of the first DU and some of its dependent DUs are already uninstalled, in that case implementations may choose to rollback and hence install those dependent DUs uninstalled in this operation. Note that the order of uninstall of DUs is implementation dependent. In any case control points depend on the evented state variables *DUIDs*, *EUIDs* and *OperationIDs*. Even in the failure case these state variables will be updated and control points will be in sync with the *ParentDevice*.

2.6.1.8 Start a Software entity successfully

1. Without Dependency

A control point invokes the *Start()* action giving the EUID of an EU to be started on an EE. After receiving the *Start()* action, the *ParentDevice* will create an operation and return the *OperationID*. This *OperationID* is added to the *OperationIDs* state variable and is evented. The operation is successfully completed once the EU corresponding to the *EUID* is started. The *EUID* is then added to the *ActiveEUIDs* list and the *RunningEUIDs* list if observed as running. The *OperationID* is removed from *OperationIDs* list. The data model MAY be updated and state variables are evented.

2. With Dependency

A control point invokes the *Start()* action giving the *EUID* of an EU to be started on an EE along with dependency handling. After receiving the *Start()* action, the *ParentDevice* will create an operation and return the *OperationID*. This *OperationID* is added to the *OperationIDs* state variable and is evented. The operation is successfully completed once the EU corresponding to the *EUID* and its dependent EUs are started. The *EUID*s are added to the *ActiveEUIDs* and *RunningEUIDs* list if observed as running. The *OperationID* is removed from *OperationIDs* list. The data model MAY be updated and state variables are evented.

2.6.1.9 Start a Software entity (failure case)

A control point invokes the *Start()* action giving the *EUID* of a first EU to be started on an EE along with dependency handling. After receiving the *Start()* action, the *ParentDevice* will create an operation and return the *OperationID*. This *OperationID* is added to the *OperationIDs* state variable and is evented. If during start an ***Error*** occurs because of a dependent EU, the start proceeds with other EUs without effecting the operation. But if an ***Error*** occurs because of the first EU and some of its dependent EUs are already started, in that case implementations may choose to rollback and hence stop those dependent EUs started in this operation. Note that the order of start of EUs is implementation dependent. The *EUID* is added to the *ErrorEUIDs* list. In any case control points depend on the evented state variables *EUIDs*, *AciveEUIDs*, *RunningEUIDs*, *ErrorEUIDs* and *OperationIDs*. Even in the failure case these state variables will be updated and control points will be in sync with the *ParentDevice*.

2.6.1.10 Stop a Software entity successfully

1. Without Dependency

A control point invokes the *Stop()* action giving the *EUID* of an EU to be stopped on an EE. After receiving the *Stop()* action, the *ParentDevice* will create an operation and return the *OperationID*. This *OperationID* is added to the *OperationIDs* state variable and is evented. The operation is successfully completed once the EU corresponding to the *EUID* is stopped. The *EUID* is then removed from the *ActiveEUIDs* list and the *RunningEUIDs* list if observed as no more running. The *OperationID* is removed from *OperationIDs* list. The data model MAY be updated and state variables are evented.

2. With Dependency

A control point invokes the *Stop()* action giving the *EUID* of an EU to be stopped on an EE along with dependency handling. After receiving the *Stop()* action, the *ParentDevice* will create an operation and return the *OperationID*. This *OperationID* is added to the *OperationIDs* state variable and is evented. The operation is successfully completed once the EU corresponding to the *EUID* along with its dependent EUs are stopped. A dependent EU will be stopped if and only if no more dependencies exist on it. The *EUIDs* are then removed from the *ActiveEUIDs* list and the *RunningEUIDs* list if they are no more observed as running. The *OperationID* is removed from *OperationIDs* list. The data model MAY be updated and state variables are evented.

2.6.1.11 Stop a Software entity (failure case)

A control point invokes the *Stop()* action giving the *EUID* of a first EU to be stopped on an EE along with dependency handling. After receiving the stop action, the *ParentDevice* will create an Operation and return the *OperationID*. This *OperationID* is added to the *OperationIDs* state variable and is evented. If during stop an ***Error*** occurs because of a dependent EU, the stop proceeds with other EUs without effecting the operation. But if an ***Error*** occurs because of the first EU and some of its dependent EUs are already stopped, in that case implementations may choose to rollback and hence start those dependent EUs stopped in this operation. Note that the order of stop of EUs is implementation dependent. The *EUID* is added to the *ErrorEUIDs* list. In any case control points depend on the evented state variables *EUIDs*, *AciveEUIDs*, *RunningEUIDs*, *ErrorEUIDs* and *OperationIDs*. Even in the failure case these state variables will be updated and control points will be in sync with the *ParentDevice*.

2.6.1.12 Firmware update

Firmware is envisioned to be a modular firmware in the context of Software Management. There are two kinds of firmware; one is the primary firmware (identified using the DUID 0 and EUID 0) and another is a normal firmware treated as any other software. The Firmware update is performed using the same Update() action which is used for software update. There are two scenarios here; one is update of primary firmware and another is update of normal firmware.

1. **Update of Primary Firmware**

A firmware with DUID 0 is already installed in the device. The corresponding EU with EUID 0 is already active. To update this primary firmware a control point will invoke Update(0, NewDUURI, 0). The following steps take place on this action invocation (see Figure 2-6):

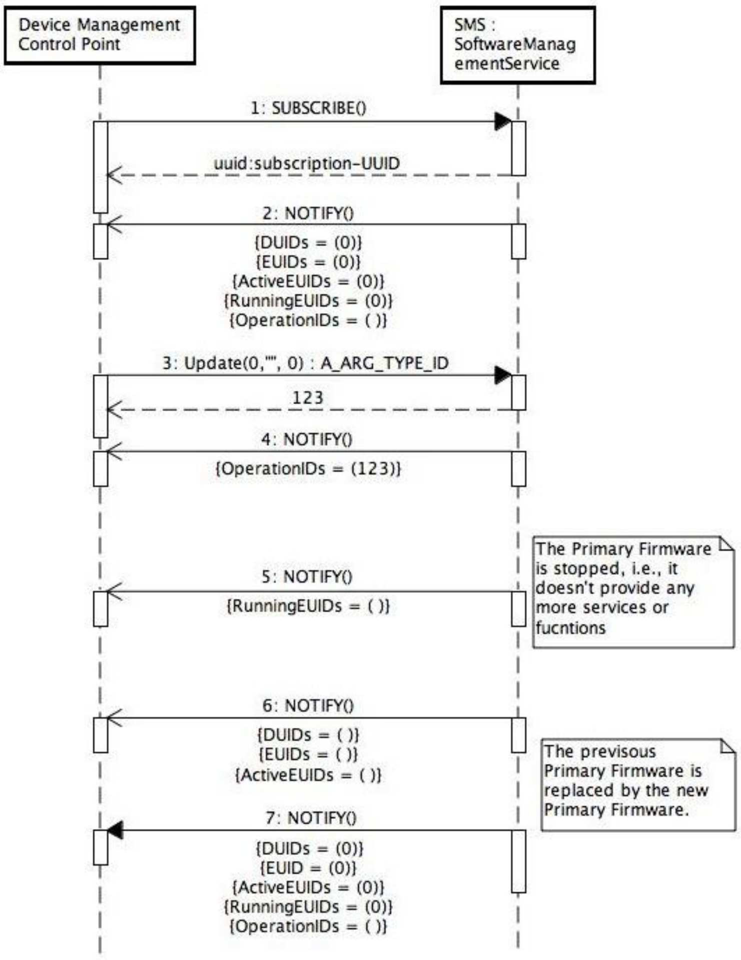


Figure 2-6 Update of Primary Firmware

DUID 0 and EUID 0 are removed from the DUIDs and EUIDs list respectively which is then evented. This is an indication to the control point that the primary firmware is getting updated.

EU 0 is stopped

DU 0 is updated

EU 0 is requested to start again

DUID 0 and *EUID* 0 is again added to the *DUIDs* and *EUIDs* list respectively which is then evented. This is an indication to the control point that the primary firmware is successfully updated.

Note: A Primary firmware MUST NOT be dependent on any other DU or EU. Hence HandleDependencies is always 0.

2. Update of Normal Firmware

Normal firmware is treated as any other software and the update procedures that are followed are similar to that of the normal software. See sections 2.6.1.4 and 2.6.1.5.

3 XML Service Description

```
<?xml version="1.0"?>
<scpd xmlns="urn:schemas-upnp-org:service-1-0">
  <specVersion>
    <major>1</major>
    <minor>0</minor>
  </specVersion>

<actionList>

    <action>
      <name>Install</name>
      <argumentList>
        <argument>
          <name>DUURI</name>
          <direction>in</direction>
          <relatedStateVariable>A_ARG_TYPE_URI</relatedStateVariable>
        </argument>
        <argument>
          <name>DUType</name>
          <direction>in</direction>
          <relatedStateVariable>A_ARG_TYPE_DUType</relatedStateVariable>
        </argument>
        <argument>
          <name>HandleDependencies</name>
          <direction>in</direction>
      <relatedStateVariable>A_ARG_TYPE_Boolean</relatedStateVariable>
        </argument>
        <argument>
          <name>OperationID</name>
          <direction>out</direction>
          <relatedStateVariable>A_ARG_TYPE_ID</relatedStateVariable>
        </argument>
      </argumentList>
    </action>

    <action>
      <name>Update</name>
      <argumentList>
        <argument>
          <name>DUID</name>
          <direction>in</direction>
          <relatedStateVariable>A_ARG_TYPE_ID</relatedStateVariable>
        </argument>
        <argument>
          <name>NewDUURI</name>
          <direction>in</direction>
          <relatedStateVariable>A_ARG_TYPE_URI</relatedStateVariable>
        </argument>
        <argument>
          <name>HandleDependencies</name>
          <direction>in</direction>
         <relatedStateVariable>A_ARG_TYPE_Boolean</relatedStateVariable>
        </argument>
        <argument>
          <name>OperationID</name>
          <direction>out</direction>
          <relatedStateVariable>A_ARG_TYPE_ID</relatedStateVariable>
        </argument>
      </argumentList>
    </action>

    <action>
      <name>Uninstall</name>
      <argumentList>
        <argument>
          <name>DUID</name>
          <direction>in</direction>
          <relatedStateVariable>A_ARG_TYPE_ID</relatedStateVariable>
        </argument>
        <argument>
          <name>HandleDependencies</name>
          <direction>in</direction>
         <relatedStateVariable>A_ARG_TYPE_Boolean</relatedStateVariable>
        </argument>
        <argument>
          <name>OperationID</name>
          <direction>out</direction>
          <relatedStateVariable>A_ARG_TYPE_ID</relatedStateVariable>
        </argument>
      </argumentList>
    </action>

    <action>
      <name>Start</name>
      <argumentList>
        <argument>
          <name>EUID</name>
          <direction>in</direction>
          <relatedStateVariable>A_ARG_TYPE_ID</relatedStateVariable>
        </argument>
        <argument>
          <name>HandleDependencies</name>
          <direction>in</direction>
         <relatedStateVariable>A_ARG_TYPE_Boolean</relatedStateVariable>
        </argument>
        <argument>
          <name>OperationID</name>
          <direction>out</direction>
          <relatedStateVariable>A_ARG_TYPE_ID</relatedStateVariable>
        </argument>
      </argumentList>
    </action>

    <action>
      <name>Stop</name>
      <argumentList>
        <argument>
          <name>EUID</name>
          <direction>in</direction>
          <relatedStateVariable>A_ARG_TYPE_ID</relatedStateVariable>
        </argument>
        <argument>
          <name>HandleDependencies</name>
          <direction>in</direction>
         <relatedStateVariable>A_ARG_TYPE_Boolean</relatedStateVariable>
        </argument>
```

```
        <argument>
          <name>OperationID</name>
          <direction>out</direction>
          <relatedStateVariable>A_ARG_TYPE_ID</relatedStateVariable>
        </argument>
      </argumentList>
    </action>

    <action>
      <name>GetDUIDs</name>
      <argumentList>
        <argument>
          <name>DUIDs</name>
          <direction>out</direction>
          <relatedStateVariable>DUIDs</relatedStateVariable>
        </argument>
      </argumentList>
    </action>

    <action>
      <name>GetEUIDs</name>
      <argumentList>
        <argument>
          <name>EUIDs</name>
          <direction>out</direction>
          <relatedStateVariable>EUIDs</relatedStateVariable>
        </argument>
      </argumentList>
    </action>

    <action>
      <name>GetActiveEUIDs</name>
      <argumentList>
        <argument>
          <name>ActiveEUIDs</name>
          <direction>out</direction>
          <relatedStateVariable>ActiveEUIDs</relatedStateVariable>
        </argument>
      </argumentList>
    </action>

    <action>
      <name>GetRunningEUIDs</name>
      <argumentList>
        <argument>
          <name>RunningEUIDs</name>
          <direction>out</direction>
          <relatedStateVariable>RunningEUIDs</relatedStateVariable>
        </argument>
      </argumentList>
    </action>

    <action>
      <name>GetDUInfo</name>
      <argumentList>
        <argument>
          <name>DUID</name>
          <direction>in</direction>
          <relatedStateVariable>A_ARG_TYPE_ID</relatedStateVariable>
        </argument>
        <argument>
          <name>DUName</name>
          <direction>out</direction>
          <relatedStateVariable>A_ARG_TYPE_Name</relatedStateVariable>
        </argument>
        <argument>
          <name>DUVersion</name>
          <direction>out</direction>

<relatedStateVariable>A_ARG_TYPE_Version</relatedStateVariable>
        </argument>
        <argument>
          <name>DUType</name>
          <direction>out</direction>
          <relatedStateVariable>A_ARG_TYPE_DUType</relatedStateVariable>
        </argument>
        <argument>
          <name>DUState</name>
          <direction>out</direction>

<relatedStateVariable>A_ARG_TYPE_DUState</relatedStateVariable>
        </argument>
        <argument>
          <name>DUURI</name>
          <direction>out</direction>
          <relatedStateVariable>A_ARG_TYPE_URI</relatedStateVariable>
        </argument>
      </argumentList>
    </action>

    <action>
      <name>GetEUInfo</name>
      <argumentList>
        <argument>
          <name>EUID</name>
          <direction>in</direction>
          <relatedStateVariable>A_ARG_TYPE_ID</relatedStateVariable>
        </argument>
        <argument>
          <name>EUName</name>
          <direction>out</direction>
          <relatedStateVariable>A_ARG_TYPE_Name</relatedStateVariable>
        </argument>
        <argument>
          <name>EUVersion</name>
          <direction>out</direction>

<relatedStateVariable>A_ARG_TYPE_Version</relatedStateVariable>
        </argument>
        <argument>
          <name>EURequestedState</name>
          <direction>out</direction>

<relatedStateVariable>A_ARG_TYPE_EURequestedState</relatedStateVariable>
        </argument>
        <argument>
          <name>EURunningState</name>
          <direction>out</direction>

<relatedStateVariable>A_ARG_TYPE_EURunningState</relatedStateVariable>
```

```
        </argument>
      </argumentList>
    </action>

    <action>
      <name>GetOperationInfo</name>
      <argumentList>
        <argument>
          <name>OperationID</name>
          <direction>in</direction>
          <relatedStateVariable>A_ARG_TYPE_ID</relatedStateVariable>
        </argument>
        <argument>
          <name>OperationState</name>
          <direction>out</direction>
<relatedStateVariable>A_ARG_TYPE_OperationState</relatedStateVariable>
        </argument>
        <argument>
          <name>Action</name>
          <direction>out</direction>
          <relatedStateVariable>A_ARG_TYPE_Action</relatedStateVariable>
        </argument>
        <argument>
          <name>ErrorDescription</name>
          <direction>out</direction>
<relatedStateVariable>A_ARG_TYPE_ErrorDescription</relatedStateVariable>
        </argument>
        <argument>
          <name>AdditionalInfo</name>
          <direction>out</direction>
          <relatedStateVariable>A_ARG_TYPE_String</relatedStateVariable>
        </argument>
      </argumentList>
    </action>

    <action>
      <name>GetOperationIDs</name>
      <argumentList>
        <argument>
          <name>OperationIDs</name>
          <direction>out</direction>
          <relatedStateVariable>OperationIDs</relatedStateVariable>
        </argument>
      </argumentList>
    </action>

    <action>
      <name>GetErrorEUIDs</name>
      <argumentList>
        <argument>
          <name>ErrorEUIDs</name>
          <direction>out</direction>
          <relatedStateVariable>ErrorEUIDs</relatedStateVariable>
        </argument>
      </argumentList>
    </action>
  </actionList>

<serviceStateTable>
    <stateVariable sendEvents="yes">
```

```
      <name>OperationIDs</name>
      <dataType>string</dataType>
    </stateVariable>

    <stateVariable sendEvents="yes">
      <name>DUIDs</name>
      <dataType>string</dataType>
    </stateVariable>

    <stateVariable sendEvents="yes">
      <name>EUIDs</name>
      <dataType>string</dataType>
    </stateVariable>

    <stateVariable sendEvents="yes">
      <name>ActiveEUIDs</name>
      <dataType>string</dataType>
    </stateVariable>

    <stateVariable sendEvents="yes">
      <name>RunningEUIDs</name>
      <dataType>string</dataType>
    </stateVariable>

    <stateVariable sendEvents="yes">
      <name>ErrorEUIDs</name>
      <dataType>string</dataType>
    </stateVariable>

    <stateVariable sendEvents="no">
      <name>A_ARG_TYPE_Boolean</name>
      <dataType>boolean</dataType>
    </stateVariable>

    <stateVariable sendEvents="no">
      <name>A_ARG_TYPE_String</name>
      <dataType>string</dataType>
    </stateVariable>

    <stateVariable sendEvents="no">
      <name>A_ARG_TYPE_ID</name>
      <dataType>ui4</dataType>
    </stateVariable>

    <stateVariable sendEvents="no">
      <name>A_ARG_TYPE_IDs</name>
      <dataType>string</dataType>
    </stateVariable>

    <stateVariable sendEvents="no">
      <name>A_ARG_TYPE_OperationState</name>
      <dataType>string</dataType>
      <allowedValueList>
        <allowedValue>Requested</allowedValue>
        <allowedValue>InProgress</allowedValue>
        <allowedValue>Completed</allowedValue>
        <allowedValue>Error</allowedValue>
      </allowedValueList>
    </stateVariable>
    <stateVariable sendEvents="no">
      <name>A_ARG_TYPE_URI</name>
      <dataType>uri</dataType>
    </stateVariable>

    <stateVariable sendEvents="no">
      <name>A_ARG_TYPE_Action</name>
      <dataType>string</dataType>
      <allowedValueList>
        <allowedValue>Install</allowedValue>
        <allowedValue>Update</allowedValue>
        <allowedValue>Uninstall</allowedValue>
        <allowedValue>Start</allowedValue>
        <allowedValue>Stop</allowedValue>
      </allowedValueList>
    </stateVariable>

    <stateVariable sendEvents="no">
      <name>A_ARG_TYPE_ErrorDescription</name>
      <dataType>string</dataType>
      <allowedValueList>
        <allowedValue>Error_None</allowedValue>
        <allowedValue>Error_ConcurrentAccess</allowedValue>
        <allowedValue>Error_MissingDependency</allowedValue>
        <allowedValue>Error_Network</allowedValue>
        <allowedValue>Error_CorruptedFile</allowedValue>
        <allowedValue>Error_DiskFull</allowedValue>
        <allowedValue>Error_Other</allowedValue>
      </allowedValueList>
    </stateVariable>

    <stateVariable sendEvents="no">
      <name>A_ARG_TYPE_DUType</name>
      <dataType>string</dataType>
      <allowedValueList>
        <allowedValue>Firmware</allowedValue>
        <allowedValue>Application</allowedValue>
        <allowedValue>Configuration</allowedValue>
        <allowedValue>Other</allowedValue>
      </allowedValueList>
    </stateVariable>

    <stateVariable sendEvents="no">
      <name>A_ARG_TYPE_DUState</name>
      <dataType>string</dataType>
      <allowedValueList>
        <allowedValue>Installing</allowedValue>
        <allowedValue>Unresolved</allowedValue>
        <allowedValue>Installed</allowedValue>
        <allowedValue>Uninstalling</allowedValue>
        <allowedValue>Uninstalled</allowedValue>
      </allowedValueList>
    </stateVariable>

    <stateVariable sendEvents="no">
      <name>A_ARG_TYPE_EURequestedState</name>
      <dataType>string</dataType>
        <allowedValue>Active</allowedValue>
        <allowedValue>Inactive</allowedValue>
    </stateVariable>
    <stateVariable sendEvents="no">
      <name>A_ARG_TYPE_EURunningState</name>
      <dataType>string</dataType>
        <allowedValue>Running</allowedValue>
        <allowedValue>Stopped</allowedValue>
        <allowedValue>Starting</allowedValue>
        <allowedValue>Stopping</allowedValue>
    </stateVariable>


  </serviceStateTable>

</scpd>
```

Appendix A Execution Platform Technologies (Informative)

This appendix is based on a first study of some execution platform technologies targeting embedded devices: Linux Debian, Microsoft .NET Platform, Java OSGi [OSGi], Java MIDP3 [MIDP]. The ongoing specification work named OMA SCOMO [SCOMO] is also studied. Like this specification, it is also an attempt to standardize a generic protocol to manage the software lifecycle of software execution platforms.

Execution Platform technologies

Most of the execution platform technologies define – at least enable – the deployment and the execution of modular software applications. After the statement of general definitions, the following technologies are described: Linux, Microsoft .NET, OSGi [OSGi], Java MIDP version 2 and 3 [MIDP], SCOMO [SCOMO].

Definitions

The management of the lifecycle of software entities is one Device Management function. The managed entities are of various kinds, e.g., applications, graphical interface items, drivers. Devices that are embedded usually define a software image that is called firmware that can only be upgraded as a whole. Other devices often provide the means to manage more fine-grained software entities that are hosted by an operating system, e.g., Windows, Linux, a modular application, e.g., a browser, or a virtual machine e.g., .NET, Java.

Execution Environment's usually define the following terms. Table 0-1 compares various software platform technologies:

- **Deployment Unit (DU)**: a binary unit that can be individually deployed on the Execution Environment. A deployment unit consists of resources such as library files, functional execution units, and configuration files, i.e. jar files, bundles, assemblies, etc.
- **Execution Unit (EU)**: a functional entity that, once started, initiates process to perform tasks or provide services, until that it is stopped. Execution units are deployed by deployment units, i.e. services, scripts, software components, MIDlets, etc.
- **Dependencies**: resource dependencies – e.g., libraries, files – can be defined between deployment units, between execution units and between deployment units and execution units. Execution units appear with the installation of deployment units. They are contained by deployment units – e.g., a .NET assembly and an OSGi bundle are deployment units that contain at most one bootstrap class that can be started, a MIDlet Suite can contain one or more MIDlets that can be started and stopped.
- **Metadata**: data that describe the software entities – vendor, update location, Execution Environment, dependencies, configuration, etc.:
    - Optional general properties
    - Platform-specific properties useful for deployment
- Software lifecycle management operations
    - For DUs, e.g., install, uninstall, update.

- For execution units, e.g., start, stop.

- Software entity states
  - Of DUs, e.g., unresolved, installed, uninstalled.
  - Of execution units, e.g., stopped, started.
- Events: they are usually related to the achievement of management operation and to state changes. Events are distinct from operation responses in that they are asynchronous and are notified to not only the operation initiator but also other managing entities.
- DU repositories: Repositories can be defined by the technology or by management tools. They are sometimes local, e.g., .NET Global Assembly Cache), and sometimes remote, e.g., RedHat RPM repositories, OSGi bundle repository.
- Execution unit repositories: Other repositories enumerate active entities, e.g., OMA SCOMO inventory, or parts of active entities, e.g., Linux process list, OSGi service registry.

Table 0-1: Execution platform technology comparisons

Technology	Deployment Unit	Execution Unit	Dependency	Actions	Events
OSGi	Bundle	Bundle	Bundle, Package, Service	DU: Install, DU: Update, DU: Uninstall EU: Start, EU: Stop	Installed, Starting, Resolved, Active, Uninstalling
Java MIDP	Midlet Suite	Midlet	Library (MIDP3)	EU: StartApp, EU: DestroyApp	NA
.NET	Assembly	Assembly	Assembly	DU: Download, DU: Load, DU: Unload (AppDomain), EU: Invoke	AssemblyLoad, AppDomainUnload, AssemblyResolve
Linux Debian	Package	RC Script, possibly others	Package	DU: PackageInstall, DU : PackageUninstall, EU : ServiceStart EU : ServiceStop	Triggering Updates
SCOMO	Delivery Package, Deployment Component	Software Component	NA	DU: Download, DU: DownloadInstall, DU: DownloadInstallInactive, DU: Install, DU: InstallInactive, DU : Update, DU : Remove, EU : Activate, EU : Deactivate	Operational Results

Linux Packages

Linux is probably the most widespread Execution Environment on the embedded devices of local networks (e.g., the home network). Linux could have been considered as a de facto standard if derivatives were not as numerous as they are today. Slackware, Debian et Red-Hat define the main families. All the other main distributions are deriving from these ones, e.g., Ubuntu, Mepis, Zenwalk, Mandriva, Suze. The structure and the lifecycle of the software entities defined by these distributions are different from one distribution to the other. However, all the distributions show some common points:

- **Package**: The Package is common to the three Linux families.

- **Execution unit**: the RC script is an entity that is common to several Linux distributions. Init processes can also be considered as execution units .
- **Dependencies**: package dependencies are explicit in some distributions like the one of the Debian family where the package manager is able to install all the packages which a package depends on when the installation of the latter is demanded. No metadata is defined to link the RC script with packages.
- **Metadata**: Processes are not described by any metadata. On the contrary, RedHat and Debian packages show metadata that describe dependencies and general deployment information. Here is the list of Debian package properties:
  - Optional general properties: Package, Version, Section, Installed-Size, Maintainer, Description.
  - Platform-specific properties useful for deployment: Priority, Architecture, Essential, Depends, Pre-Depends, Recommends, Suggests, Conflicts, Replaces, Provides.
- Software lifecycle management operations (see Figure)
  - For DUs: Install, Remove, Upgrade[2].
  - For execution units: Start, Stop. They are the most common operations that can be performed on RC scripts or other init processes .
- Software entity states (see Figure)
  - Of Deployment Units: Installed, Removed, Resolved. Since dependency resolution mechanisms are available on Linux Debian and RedHat platforms, the "Resolved" state is a state that can be displayed for the DUs. Many transitory states are specified but they are not represented here.
  - Of Execution Units: Inactive, Active. RC scripts can be defined whereas useful DUs are not available yet. However, they can only be started when the necessary DUs are installed.
- Events: It is possible to be notified of the activity change of the RC scripts thanks to the process table. POSIX signals are also available for the follow-up of init daemons.
- DU repositories: Debian and RedHat repositories.
- Execution unit repositories: The process table enumerates the running and defunct processes.

---

[2] Even if the detailed Debian manual is more sophisticated, the overall set of operations is about installation, update and uninstallation: http://www.debian.org/doc/debian-policy/ch-maintainerscripts.html#s-mscriptsinstact

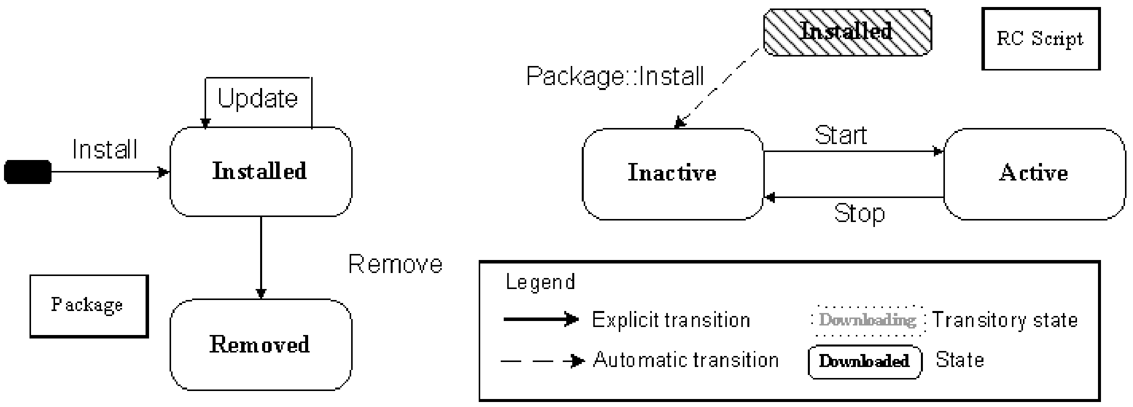


Figure 0-1 A general vision of Linux software entities and their state diagrams

OSGi bundles

The OSGi technology defines a modular deployment platform for Java applications. Far from the initial vision of an Internet Gateway at creation time in 1999, a more general objective has been targeted by the OSGi Alliance: "OSGi technology is the dynamic module system for Java" is written on their web site. The technology defines fine-grained code sharing and isolation mechanisms between deployment units called bundles. These mechanisms and a set of good practices make it possible to simultaneously execute applications constituted by several bundles that coexist and share code and loaded objects. Sharing and isolation rules are applied by the platform according to bundle dependencies described in their manifest and security directives.

Here are the characteristics of the bundle lifecycle defined by the OSGi specification release 4 version 4.1 [OSGi]:

- Deployment Unit: the OSGi bundle is a JAR file that may contain Java class files, other JAR files, a Manifest file describing bundle metadata, other non-OSGi resources.
- Execution Unit: Every OSGi bundle declaring a fine named "Activator" can be started or stopped on the platform. The other bundles are inert resource libraries.
- Dependencies: Every OSGi bundle must declare the versioned packages it provides, the ones it requires and the ones that remain private inside the bundle. It may also define dependencies towards bundles. The OSGi platform is responsible for the resolution diagnostic of bundle package dependencies. A non resolved bundle is prevented to start by the platform. Some more sophisticated component model (e.g., OSGi Declarative Services) enable the definition of internal components inside bundles and service dependencies between them.
- Metadata: The following properties are defined for OSGi bundles:
    - Optional general properties: Bundle-Category, Bundle-ContactAddress, Bundle-Copyright, Bundle-Description, Bundle-DocURL, Bundle-Localization, Bundle-Name, Bundle-Vendor, Bundle-Version.
    - Platform-specific properties useful for deployment: Bundle-ActivationPolicy, Bundle-Activator, Bundle-Classpath, Bundle-ManifestVersion, Bundle-NativeCode, Bundle-RequiredExecutionEnvironment, Bundle-SymbolicName, Bundle-UpdateLocation, DynamicImport-Package, Export-Package, Export-Service, Fragment-Host, Import-Package, Import-Service, Require-Bundle.

- Software lifecycle management operations (see Figure 0-2): Every bundle can be installed, uninstalled, updated, started, stopped thanks to the available methods: Install, Uninstall, Update, Start, Stop. However, starting and stopping a bundle that does not declare an "Activator" file have no effect. Despite a "Resolved" state is specially specified, resolution mechanisms can only be triggered by the platform itself (no "Resolve" method is defined).
- Software entity states (see Figure 0-2): Installed, Resolved, Uninstalled, Active. The "Resolved" state indicates that the resolution diagnostic has been performed and that required packages (and bundles) are effectively available on the platform. OSGi bundles enter transitory states when start and stop operations are initiated: Starting, Stopping.
- Events: "Installed", "Uninstalled", "Updated", "Resolved", "Unresolved", "Starting", "Started", "Stopping", "Stopped". The bundle activity is also visible thanks to the state notification of provided services: "Registered", "Modified" and "Unregistered".
- Deployment Unit repositories: The de facto OBR standard (OSGi Bundle Repository) describes a list of stored bundles. This format can be used not only for remote repositories but also for local ones.
- Execution unit repositories: Well-known OSGi platforms provide platform administrators with a tool listing the hosted OSGi bundles with their state and metadata. The tool also enables operations on the bundles. Services and their states may also be listed.

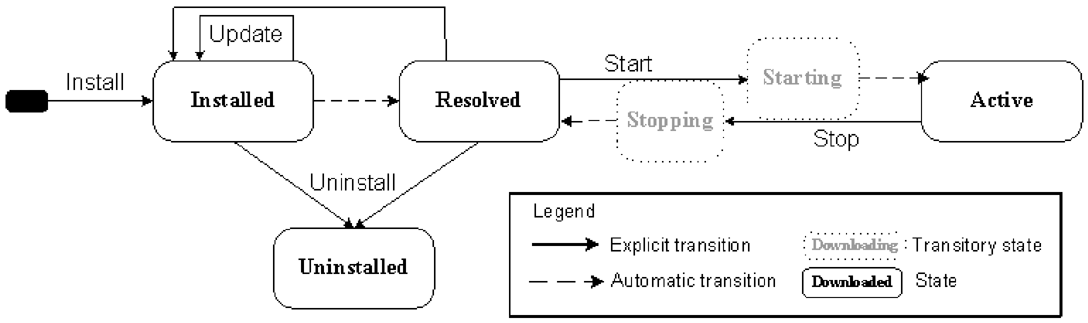


Figure 0-2 The OSGi bundle lifecycle

Java MIDlets

Java MIDP (Mobile Information Device Profile) defines software entities deployable on a constrained Java virtual machine specified by Java CLDC (Connected Limited Device Configuration). MIDP is known under 3 versions. What is indicated here is true on MIDP2 [MIDP] and on MIDP3. Much more restrictive than the OSGi model, CLDC/MIDP specifies an all-or-nothing mode for code sharing between software entities. This mode follows the "sandbox" model. Inside a MIDlet Suite – the MIDP deployment unit, every part has access to the other parts and to the librairies hosted by the underlying platform. However, no part has access to the code inside other MIDlet Suites. If MIDP3 brings the notion of library (LIBlet) sharing between MIDlet Suites, this follows the sandbox model by preventing the sharing of objects at runtime (the shared library is separately loaded by each requesting MIDlet Suite). Only Inter-Process Communication (IPC) is enabled between MIDlet Suites. Every MIDlet Suite may contain several execution units called MIDlets. The lifecycle of each of these entities is described on Figure 0-3.

Here are the platform characteristics [MIDP]:

- Deployment Unit: a MIDlet Suite is a JAR file containing the code of one or several MIDlets, a JAD (Java Application Descriptor) file describing them and some non-Java resources.

- Execution Unit: MIDlet.
- Dependencies: A MIDlet Suite may contain one or several MIDlets. Before MIDP3, dependencies were defined neither between MIDlets nor between MIDlet Suites. With MIDP3, every MIDlet may provide and require inert libraries.
- Metadata : les métadonnées sont écrites dans le fichier JAD.
  - Optional general properties: MIDlet-Name, MIDlet-Version, MIDlet-Vendor, MIDlet-Jar-URL, MIDlet-Jar-Size, MIDlet-Description, MIDlet-Icon, MIDlet-Info-URL, MIDlet-Data-Size.
  - Platform-specific properties useful for deployment: MicroEdition-Profile, MicroEdition-Configuration, MIDlet-n, MIDlet-Install-Notify, MIDlet-Delete-Notify, MIDlet-Permissions, MIDlet-Permissions-Opt, MIDlet-Push-n, MIDlet-specific attributes, MIDlet-Jar-RSA-SHA1
- Software lifecycle management operations (see Figure 0-3)
  - For Deployment Units: Install, Remove, Update.
  - For Execution Units: StartApp, StopApp. The first two versions of MIDP specified a pausing operation (PauseApp). This operation is now deprecated in the 3rd version.
- Software entity states (see Figure 0-3)
  - Of Deployment Units: Installed, Removed.
  - Of Execution Units: Destroyed, Active. The "Paused" state is deprecated in MIDP3. MIDlets appear with the "Destroyed" state when the associated MIDlet Suite is installed (see on the right of Figure 0-3).
- Events: Only a global application manager, called JAM (Java Application Manager), is aware of state transition of MIDlets.
- Deployment Unit repositories: It is possible to create a MIDlet Suite repository thanks to the information of MIDlet Suite application descriptors (JAD files).
- Execution unit repositories: MIDP platforms usually provide a tool listing available MIDlets and their state.

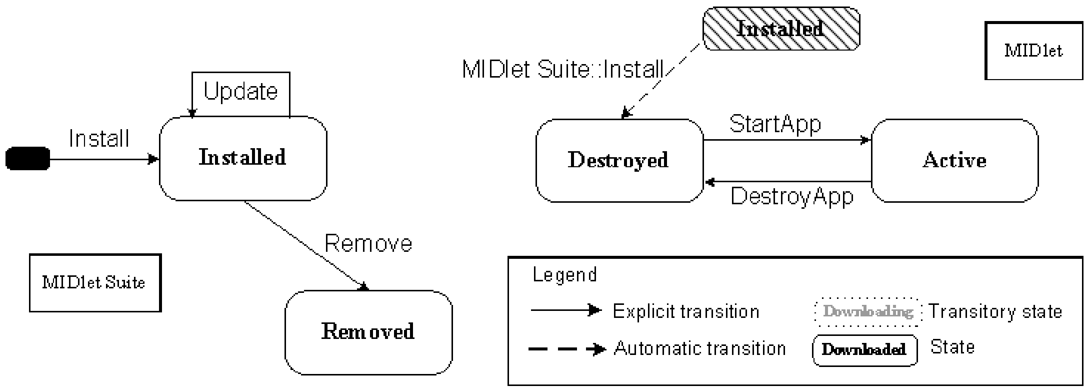


Figure 0-3 The lifecycle of MIDlet Suites and the one of their MIDlets

.NET Assemblies

.NET is an Execution Environment specified by Microsoft. The innovation mainly relies in the definition of a virtual machine interpreting a binary code that is common to the compilation of several programming languages (e.g., C#, VB#, J#). .NET is born in the same period as the OSGi standard (see section 0). Some similar characteristics are shared by these technologies: virtual machine, dynamic loading of software software entities called "assembly", assembly dependency resolution performed by the underlying platform thanks to assembly metadata. Code sharing and isolation are defined at the type level. Visibility levels called "public" and "internal" distinguish the types that are visible outside the assembly from the types that remain private to the assembly internal code. The public assembly code is visible from the other assemblies hosted by the same Application Domain (AppDomain) whereas the assemblies of distinct Domains can only communicate through the Inter-Process Communication (IPC). An assembly can be dynamically, i.e., at runtime, loaded by one or several domains. However, the assemblies are separately loaded by distinct Domains and one assembly can not be unloaded by a domain without unloading the whole domain. These limits make the .NET modularity appear between the OSGi flexibility and the all-or-nothing mode of the MIDP3 sandbox model.

Here, the .NET platform characteristics are summarized:

- Deployment Unit: An assembly is a set of resources described by a manifest file. It can be dynamically loaded by one or several Application Domains.
- Execution unit: Every activable assembly offers an Entry Point with a generic invocation method, named invoke(), accepting a table of input arguments.
- Dependencies: Every assembly declares public types and the assemblies that it requires.
- Metadata: The following properties can be found in the manifest file of an assembly. For convenience, the "Assembly" prefix of every property name has been removed in the list:
  - Optional general properties: Culture, Flags, Version, Company, Copyright, FileVersion, InformationalVersion, Product, Trademark, DefaultAlias, Description, Title.
  - Platform-specific properties useful for deployment: Name, FileList, TypeReference, ReferencedAssemblies, EntryPoint, Configuration, DelaySign, KeyFile, KeyName.
- Software lifecycle management operations (see Figure 0-4) : every assembly can be
  - Downloaded and removed in the local repository called "Global Assembly Cache" or in the AppDomain particular caches.
  - Then loaded (Assemly.Load()), invoked (Assembly.EntryPoint.invoke(Object[] optionalParams) or AppDomain.ExecuteAssembly(Assembly a)). Dependency resolution is performed by the underlying platform itself.
- Software entity states (see Figure 0-4): Downloaded, Loaded, Unloaded. No activity state is defined. And although resolution mechanisms are part of the platform, the successful invocation (activation) of an assembly does not prevent the hosting Domain to throw "AssemblyResolve" exceptions when executing it.
- Events: "AssemblyLoad" when an assembly is loaded, "AppDomainUnload" when a Domain is unloaded, "AssemblyResolve" when a required type is missing at the moment of use.
- Deployment Unit repositories: The Global Assembly Cache is a local assembly cache where applications share assemblies. Every Domain has also a private cache.
- Execution unit repositories: The .NET platform is delivered on Windows with some tools like the .Net Framework Configuration, which maintains the list of available assemblies and enables their removal.

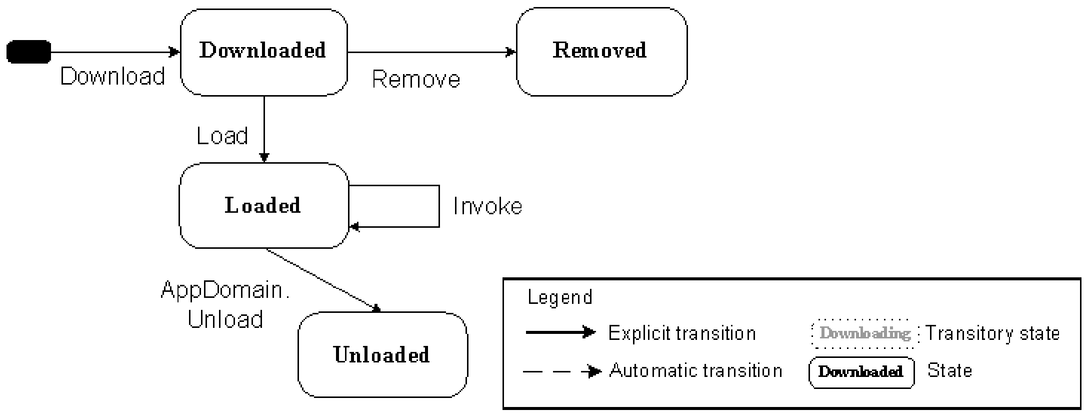


Figure 0-4 The state diagram of a .NET assembly

SCOMO Components: a attempt of generalization

SCOMO [SCOMO] is an ongoing specification of the Open Mobile Alliance. Unlike the previous execution platforms, SCOMO is not a software platform associated to a programming language. It is a protocol, which, linked to OMA DM protocols, enables the software management of software entities of various embedded platforms. It then follows a general approach similar to the one of the specification of the Software Management Service in the UPnP Device Management Working Committee, i.e., the definition of a generic protocol for the management of any software platform technology. SCOMO Delivery Package and Deployment Component both part of the DU concept while SCOMO Software Component matches the EU concept of this specification.

Here are the "platform" characteristics:

- Deployment Unit: The concept is specified into two separate entities: the SCOMO Delivery Package (DP) and the SCOMO Deployment Component (DC). After a DP is downloaded, the set up of it installs the contained DCs. The removal of a DP does not involve the removal of the related DCs. A DC can be explicitly removed by a command. Activation and Deactivation of a DC are defined but these operations are not similar to the activation and deactivation of an EU in this specification. Indeed, the active DCs are not running. The activation only makes "services or resources it embodies accessible to other entities or resources (including end-user)" so that "a service that consists of multiple Deployment Components is ready for launch after all the relevant Deployment Components have reached the Active state" (see [SCOMO])[3].
- Execution Unit: The concept is reified into SCOMO Software Component (SC). However, SCOMO does not define any operation on SCs and leaves the launch of "services" to terminal-specific interactions.

[3] Moreover, the inactive state matches more the Unresolved state of the UPnP specification than the Inactive one. Indeed, this state prevents resources to be accessed by other software entities: "The main goal of Inactive state is to minimise the downtime of Deployment Component management operations. Interference with external events (e.g. some end-user actions) could disturb or even block some management tasks. Inactive state is a powerful concept when implementing fault tolerant systems it enables controlled management operations. e.g. safe removals of Deployment Components from runtime-environment."

- Dependencies: A DP may contain one or several DCs. A DC contains a Software Component and related metadata.
- Metadata: Every entity is defined with metadata that are described in a device hierarchical data model. Several metadata items describing a DP are identical to the ones describing a DC:
  - Optional general properties: name, description, version, PkgType.
  - Platform-specific properties useful for deployment: PkgID (DP), PkgURL, ID (DC), data.
- Software lifecycle management operations:
  - For DPs (see Figure 0-5): Download, DownloadInstall, DownloadInstallInactive, Install, InstallInactive, Remove. Some primitives are defined as "composed" in SCOMO specification: DownloadInstall, DownloadInstallInactive. The rationale may be to better match the operations defined in various software platforms or define shortcuts in order to speed up the networked process. The following citation seems in favour of the second reason: "When a Composed Primitive is executed, two state transitions happen in the Device. For example if DownloadInstall is executed, a Deployment Component transits from Not Downloaded State to Delivered State after successful download procedure. It transits to Active State after successful installation procedure. If the latter processing fails it remains in previous state and the second state transition does not happen."
  - For DCs (see Figure 0-6): Activate, Deactivate, Remove. Deployment units can be explicitly activated or deactivated in this specification. The Deactivated state represents a state where no applications can use the DC (similarly to the UPnP SMS Unresolved state).. The explicit activation makes the contained services and resources accessible to applications and the end-user. The SCOMO activation may correspond to an explicit resolution action in the UPnP SMS diagrams with EUs startable only when the DU becomes resolved (Installed state).
  - For SCs: No operation is defined.
- Software entity states:
  A. Of DPs (see Figure 0-5): Not Downloaded, Delivered, Installed, Removed. Quoted in the specification: "Delivered State enables “deliver-first-install-later”-like use cases e.g. updates of mobile office solutions requiring all the components to be activated immediately after back-end update. In this case the Delivery and Deployment are discrete".

  B. Of DCs (see Figure 0-6): Inactive, Active, Removed.

  C. Of SCs: No state is defined.
- Events: Alerts are defined for every state change.
- Deployment Unit repositories: The inventory of DPs in the "Not Downloaded" state lists the DPs that are available for delivery. The inventory of DPs in the "Delivered" state lists the DPs that are available for installation.
- Execution Unit repositories: A DC inventory is also specified in the device data model.

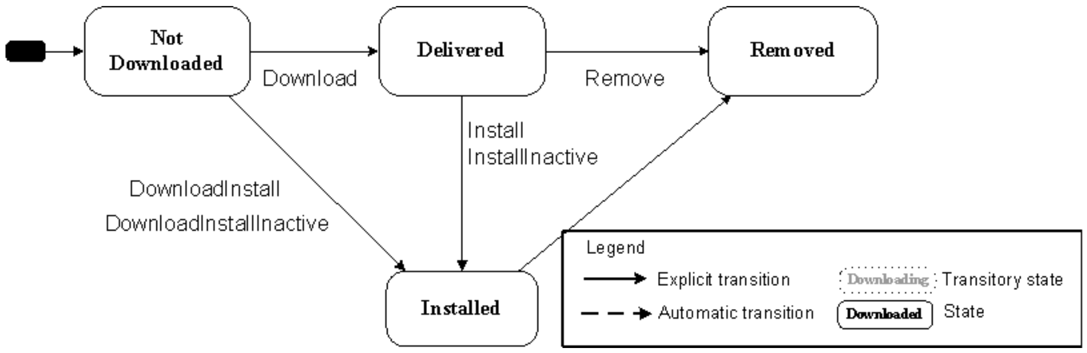


Figure 0-5 SCOMO Delivery Package State Diagram

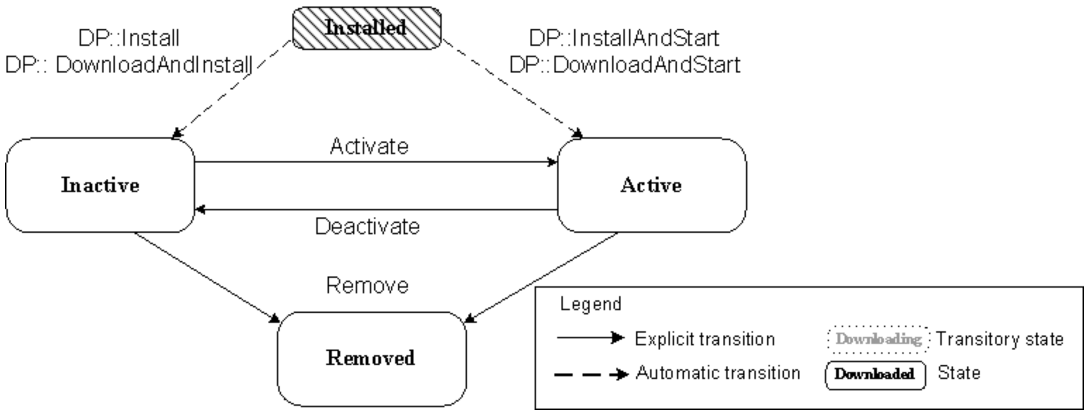


Figure 0-6 SCOMO Deployment Component State Diagram

Appendix B Software Data Model (Normative)

Software Data Model is Optional for any SMS implementation. Implementations which choose to provide additional information to control points can implement *Software Data Model* and support CMS to access this *Software Data Model*. Refer CMS [CMS] document for notations used in the below tables.

Name	**Type**	**Acc**	**Req**	**Description**	**EoC**	**Ver**
/UPnP/DM/Software/	*SingleInstance*	-	R	This node contains general information on a DU instance.	-	-
DUNumberOfEntries	unsignedInt	-	R	Number of DU instances.	-	-
/UPnP/DM/Software/ Capabilities/	*SingleInstance*		R	Information related to software capabilities of the targeted Execution Environment.	-	-
AbleToAutoStart	boolean	-	R	Indicates the ability to automatically start an EU at start time.	1	R
AbleToAutoRestart	boolean	-	R	Indicates the ability to automatically restart an EU.	1	R
AbleToHandleDUDep endency	boolean	-	R	Indicates the ability to handle dependency when a DU is installed or uninstalled.	1	R
AbleToHandleEUDep endency	boolean	-	R	Indicates the ability to handle dependency when an EU is started or stopped.	1	R
/UPnP/DM/Software/ DU/#/	*MultiInstance*	-	R	This node contains general information of all the DUs	-	-
DUID	unsignedInt	-	R	DU identifier Unique key. DUID 0 is reserved to identify the primary firmware (see section 2.2.4.	-	-
State	string	-	R	State of the DU (see section 2.2.3.1) Allowed values are: "Installing" "Unresolved" "Installed" "Uninstalling" "Uninstalled"	1	R
Type	string	-	R	Type of the resources contained in the DU Allowed values are defined in SMS specification. See Table 2-5.	-	-
Name	string	-	R	Name of the DU. For example the name of the DU can be the one used by the targeted EE.	-	-
Description	string	-	R	Textual description of the DU. Format of this value is EE specific	-	-
Version	string	-	R	Version of the DU. Format of this value is EE specific.	1	R
URI	string	-	R	URI of the DU. May be used in *Update()* when the NEWDUURI argument is not specified	1	R
SystemID	string	-	R	Internal ID. Format of this value is EE specific. The value should be the ID assigned by the EE to the DU	-	-

Name	Type	Acc	Req	Description	EoC	Ver
Size	unsignedInt	-	O	Size in Byte of the deployment unit when installed. This value is not intended to vary during the lifetime of the DU. Nevertheless it may be updated when a new version of the DU is installed.	-	-
Date	dateTime		O	Date of installation or of the last update of the DU.	-	-
Dependencies	string		O	CSV list of DUIDs that the DU depends on. An empty value should be interpreted as: the current instance does not depend on other DUs.	-	-
EUNumberOfEntries	unsignedInt	-	R	Number of EU instances contained in this DU.	-	-
/UPnP/DM/Software/DU/#/EU/#/	*MultiInstance*		R	This node contains general information on a particular EU instance.	-	-
EUID	unsignedInt	-	R	EU identifier provided by the ParentDevice on DU installation. Unique key, also unique across all the DUs. EUID 0 is reserved to identify the primary firmware (see section 2.2.4).	-	-
RequestedState	string		R	*Requested state* of the EU (see section 2.2.3.2) Allowed values are: "Active" "Inactive" Active means that the *Start()* action has been called to make this EU running. Inactive means that the *Stop()* action has been called to prevent this EU to be executed.	1	R
RunningState	string	-	R	Current *Running State* of the EU (see section 2.2.3.2). Allowed values are: "Running" "Stopped" "Starting" "Stopping"	1	R
Name	string	-	R	EU friendly name. Could be used to track the EU activity in the EE. Format of this value is EE specific	-	-
Version	string	-	O	Version of the EU. Format of this value is EE specific. If this DU is the primary firmware (DUID=0), this value MAY be equal to /UPnP/DM/DeviceInfo/SoftwareVersion	1	R
Description	string	-	R	Textual description of the EU. Format of this value is EE specific	-	-
AutoStart	boolean	W	O	Indicate that the EU has to be automatically started at start time. The initial value is provided by the DU when installing.	1	R
AutoRestart	boolean	W	O	Indicate that the EU has to be automatically re-started each time it is stopped by another means that the call to the *Stop()* action. The initial value is provided by the DU when installing.	1	R

Name	Type	Acc	Req	Description	EoC	Ver
SystemID	string	-	R	Internal ID provided by the EE while the EU is in the *Active* state. Format of this value is EE specific. The value should be the ID assigned by the EE to the EU.	-	-
Dependencies	string		R	CSV list of EUIDs that the EU depends on. An empty value should be interpreted as: the current instance does not depend on other EUs.	-	-
SystemPath	string		O	System path to the EU's bootstrap.	-	-
Error	String		O	Detailed description of EU Errors	-	-