
S60 2nd Edition: Implementing Device Management Plug-in Adapter

Version 1.1
November 15, 2006

S60 platform

Legal notice

Copyright © 2005–2006 Nokia Corporation. All rights reserved.

Nokia and Forum Nokia are registered trademarks of Nokia Corporation. Other product and company names mentioned herein may be trademarks or trade names of their respective owners.

Disclaimer

The information in this document is provided “as is,” with no warranties whatsoever, including any warranty of merchantability, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. Furthermore, information provided in this document is preliminary, and may be changed substantially prior to final release. This document is provided for informational purposes only.

Nokia Corporation disclaims all liability, including liability for infringement of any proprietary rights, relating to implementation of information presented in this document. Nokia Corporation does not warrant or represent that such use will not infringe such rights.

Nokia Corporation retains the right to make changes to this specification at any time, without notice.

License

A license is hereby granted to download and print a copy of this specification for personal use only. No other license to any other intellectual property rights is granted herein.

Contents

1.	Introduction	5
2.	Implementing interface functions	6
2.1	NewL	6
2.2	DDFVersionL.....	6
2.3	DDFStructureL	6
2.4	AddLeafObjectL	10
2.5	UpdateLeafObjectL	11
2.6	DeleteObjectL	11
2.7	FetchLeafObjectL.....	12
2.8	ChildURIListL	12
2.9	AddNodeObjectL.....	13
2.10	LuidMappingInAdapterL.....	14
2.11	AdapterCanChangeACL	14
2.12	UpdateACLPropertyL.....	15
2.13	EndMessageL	15
3.	Terms and abbreviations	17
4.	References	18
5.	Evaluate this resource	19

Change history

July 15, 2005	Version 1.0	Initial document release
November 15, 2006	Version 1.1	Minor updates to Chapter 1 and 4. Document title updated (earlier title was <i>How To Implement The DM Plug-in Adapter</i>).

1. Introduction

This document gives step-by-step instructions on how to implement a new device management (DM) plug-in adapter for S60 2nd Edition. The plug-in interface is described in *API Reference* [1] and all the DM plug-in adapters implement the same interface.

The ECOM architecture is used for loading and using the adapters. The ECOM architecture is part of the Symbian SDK and documentation of it can be found from the Symbian documentation [2].

Basically, implementing a new DM plug-in adapter has the following two steps:

- Creating an ECOM resource file and implementing the functions needed by the ECOM framework. The DM plug-in interface uid is defined in the `nsmlmadapter.h` header file. (`KNSm1DMInterfaceUid = 0x101F4D3B`)
- Inherit the new adapter from the `CNSm1DmAdapter` class and implement the virtual functions. The `CNSm1DmAdapter` class is defined in the `nsmlmadapter.h` header file. The pure virtual functions must be implemented in every adapter. There are also functions that have a default implementation and adapters do not necessarily need to implement them. The following chapter gives more detailed descriptions of the functions.

Note that in S60 3rd Edition the DM APIs have changed. For the header files and documentation on implementing the device management plug-in for S60 3rd Edition, refer to the [S60 3rd Edition: Device Management Plug-in](#) [4] that is available in Forum Nokia.

2. Implementing interface functions

The following sections describe the implementations of the DM adapter interface functions. The section titles are the names of the functions in the interface, that is, the functions that are inherited from the `CNSm1DmAdapter` class.

2.1 NewL

Prototype:

```
static CNSm1DmAdapter* NewL(MNSm1DmCallback* aDmCallback);
```

The function creates an instance of a newly created adapter and returns a pointer to it. `MNSm1DmCallback* aDmCallback` is the callback interface for giving status codes and results when the real commands are executed (add, replace, delete, or fetch). Because of this, the pointer to the callback interface should be taken to a member variable for later use. The `MNSm1DmCallback` interface is explained in *API Reference* [1].

Example code:

```
CNSm1DmExmplAdapter* CNSm1DmExmplAdapter::NewL(MNSm1DmCallback*
aDmCallback )
{
    CNSm1DmExmplAdapter* self = new (ELeave) CNSm1DmSctsAdapter();
    self->iCallback = aDmCallback;
    return self;
}
```

2.2 DDFVersionL

Prototype:

```
void DDFVersionL( CBufBase& aVersion );
```

The function fills the version of the adapter's Device Description Framework (DDF) to `aDDFVersion`. The version is used for checking if the DDF of a device has been changed, that is, the version must be updated if a new version of the adapter has been made and the DDF has changed.

Example code:

```
void CNSm1DmExmplAdapter::DDFVersionL(CBufBase& aDDFVersion)
{
    aDDFVersion.InsertL(0, _L8("1.0"));
}
```

2.3 DDFStructureL

Prototype:

```
void DDFStructureL(MNSm1DmDDFObject& aDDF);
```

This function is used for telling the DDF structure to the engine module. The engine module gives a reference to the root node of the DDF, and the DDF structure of the adapter is built on that. The DDF building is started by calling `AddChildObjectL` to `aDDF`. The function call creates a new child node under the root node and returns a reference to it. The newly created node implements the `MNSm1DmDDFObject` interface and the properties of the node are set by calling setter functions that are defined in *API*

Reference [1]. The building of the DDF structure must be done carefully, because the engine module uses the DDF for directing the commands to the correct adapter and also checks the supported commands from the DDF. Mistakes in here may, therefore, cause that the commands will never come to the adapter correctly.

The correct adapter is identified from the beginning of the URI. For that, there is the `SetAsObjectGroupL` function in the `MNSm1DmDDFObject` interface. This is called for nodes that are part of the URI that identifies the adapter. For example, in case of “`SyncML/DMAcc/<x>`”, the `SyncML` and `DMAcc` nodes are object group nodes and in case of e-mail settings, there is only one object group node, that is, the `Email` node. So, commands where the URI starts with `SyncML/DMAcc` are directed to the DM settings adapter and commands where the URI starts with `Email` are directed to the e-mail settings adapter, and so on.

The URI validity is also checked: if the URI start is OK but the rest of the URI is against the DDF structure, the command will never come to the adapter.

The name of the node is given when creating the node, that is, when calling the `AddChildObjectL`. In case of dynamic nodes, the name **MUST** be an empty string.

The access types are set by creating a new instance of `TNSm1DmAccessTypes`, setting the wanted access types to it, and calling the `SetAccessTypesL` of the newly created node.

The example code below defines the DDF structure shown in Figure 1.

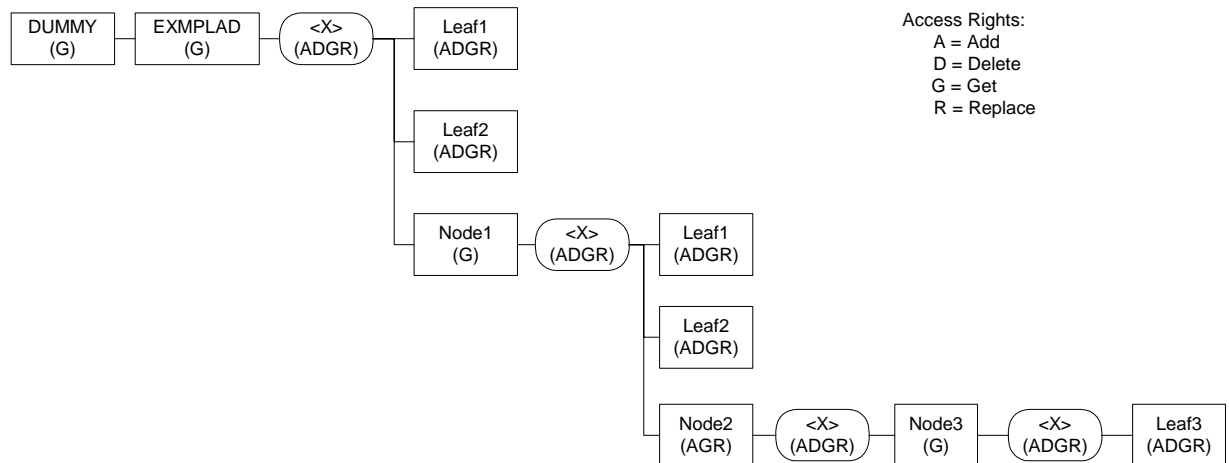


Figure 1: DDF structure of the example adapter

Example code:

```

void CNSm1DmExmplAdapter::DDFStructureL( MNSm1DmDDFObject& aDDF )
{
    TNSm1DmAccessTypes accessTypesAll;
    accessTypesAll.SetGet();
    accessTypesAll.SetDelete();
    accessTypesAll.SetAdd();
    accessTypesAll.SetReplace();

    TNSm1DmAccessTypes accessTypesGet;
    accessTypesGet.SetGet();

    TNSm1DmAccessTypes accessTypesNoDelete;
    accessTypesNoDelete.SetGet();
    accessTypesNoDelete.SetAdd();
    accessTypesNoDelete.SetReplace();
}
  
```

```

// DUMMY
MNSmlDmDDFObject& dummy = aDDF.AddChildObjectL(_L("DUMMY"));
dummy.SetAccessTypesL(accessTypesGet);
dummy.SetAsObjectGroup();
dummy.SetOccurrenceL(MNSmlDmDDFObject::EOne);
dummy.SetScopeL(MNSmlDmDDFObject::EPermanent);
dummy.SetDFFFormatL(MNSmlDmDDFObject::Enode);

// EXMPLAD
MNSmlDmDDFObject& exmplAd = dummy.AddChildObjectL(_L("EXMPLAD"));
exmplAd.SetAccessTypesL(accessTypesGet);
exmplAd.SetAsObjectGroup();
exmplAd.SetOccurrenceL(MNSmlDmDDFObject::EOne);
exmplAd.SetScopeL(MNSmlDmDDFObject::EPermanent);
exmplAd.SetDFFFormatL(MNSmlDmDDFObject::Enode);

//run-time node
MNSmlDmDDFObject& rtAcc = exmplAd.AddChildObjectL(KNullDesC);
rtAcc.SetAccessTypesL(accessTypesAll);
rtAcc.SetOccurrenceL(MNSmlDmDDFObject::EZeroOrMore);
rtAcc.SetScopeL(MNSmlDmDDFObject::EDynamic);
rtAcc.SetDFFFormatL(MNSmlDmDDFObject::Enode);

//leaf1
MNSmlDmDDFObject& leaf1 = rtAcc.AddChildObjectL(_L("Leaf1"));
leaf1.SetAccessTypesL(accessTypesAll);
leaf1.SetOccurrenceL(MNSmlDmDDFObject::EOne);
leaf1.SetScopeL(MNSmlDmDDFObject::EDynamic);
leaf1.SetDFFFormatL(MNSmlDmDDFObject::Echr);
leaf1.AddDFFTypeMimeTypeL(KNSmlDMSctsTextPlain);

//leaf2
MNSmlDmDDFObject& leaf2 = rtAcc.AddChildObjectL(_L("Leaf2"));
leaf2.SetAccessTypesL(accessTypesAll);
leaf2.SetOccurrenceL(MNSmlDmDDFObject::EOne);
leaf2.SetScopeL(MNSmlDmDDFObject::EDynamic);
leaf2.SetDFFFormatL(MNSmlDmDDFObject::Echr);
leaf2.AddDFFTypeMimeTypeL(KNSmlDMSctsTextPlain);

// Node1
MNSmlDmDDFObject& node1 = rtAcc.AddChildObjectL(_L("Node1"));
node1.SetAccessTypesL(accessTypesGet);
node1.SetOccurrenceL(MNSmlDmDDFObject::EOne);
node1.SetScopeL(MNSmlDmDDFObject::EPermanent);
node1.SetDFFFormatL(MNSmlDmDDFObject::Enode);

// rt2
MNSmlDmDDFObject& rt2 = node1.AddChildObjectL(KNullDesC);
rt2.SetAccessTypesL(accessTypesAll);
rt2.SetOccurrenceL(MNSmlDmDDFObject::EOne);
rt2.SetScopeL(MNSmlDmDDFObject::EPermanent);
rt2.SetDFFFormatL(MNSmlDmDDFObject::Enode);

//leaf11
MNSmlDmDDFObject& leaf11 = rt2.AddChildObjectL(_L("Leaf1"));
leaf11.SetAccessTypesL(accessTypesAll);
leaf11.SetOccurrenceL(MNSmlDmDDFObject::EOne);
leaf11.SetScopeL(MNSmlDmDDFObject::EDynamic);
leaf11.SetDFFFormatL(MNSmlDmDDFObject::Echr);
leaf11.AddDFFTypeMimeTypeL(KNSmlDMSctsTextPlain);

//leaf12
MNSmlDmDDFObject& leaf12 = rt2.AddChildObjectL(_L("Leaf2"));
leaf12.SetAccessTypesL(accessTypesAll);
leaf12.SetOccurrenceL(MNSmlDmDDFObject::EOne);
leaf12.SetScopeL(MNSmlDmDDFObject::EDynamic);
leaf12.SetDFFFormatL(MNSmlDmDDFObject::Echr);

```

```

leaf12.AddDfTypeMimeTypeL(KNSmldMSctsTextPlain);

// Node2
MNSmldmDDFObject& node2 = rtAcc.AddChildObjectL(_L("Node2"));
node2.SetAccessTypesL(accessTypesGet);
node2.SetOccurrenceL(MNSmldmDDFObject::EOne);
node2.SetScopeL(MNSmldmDDFObject::EPermanent);
node2.SetDffFormatL(MNSmldmDDFObject::Enode);

// rt3
MNSmldmDDFObject& rt3 = node2.AddChildObjectL(KNullDesC);
rt3.SetAccessTypesL(accessTypesAll);
rt3.SetOccurrenceL(MNSmldmDDFObject::EOne);
rt3.SetScopeL(MNSmldmDDFObject::EPermanent);
rt3.SetDffFormatL(MNSmldmDDFObject::Enode);

// Node3
MNSmldmDDFObject& node3 = rt3.AddChildObjectL(_L("Node3"));
node3.SetAccessTypesL(accessTypesGet);
node3.SetOccurrenceL(MNSmldmDDFObject::EOne);
node3.SetScopeL(MNSmldmDDFObject::EPermanent);
node3.SetDffFormatL(MNSmldmDDFObject::Enode);

// rt4
MNSmldmDDFObject& rt4 = node3.AddChildObjectL(KNullDesC);
rt4.SetAccessTypesL(accessTypesAll);
rt4.SetOccurrenceL(MNSmldmDDFObject::EOne);
rt4.SetScopeL(MNSmldmDDFObject::EPermanent);
rt4.SetDffFormatL(MNSmldmDDFObject::Enode);

//leaf3
MNSmldmDDFObject& leaf3 = rt4.AddChildObjectL(_L("Leaf3"));
leaf3.SetAccessTypesL(accessTypesAll);
leaf3.SetOccurrenceL(MNSmldmDDFObject::EOne);
leaf3.SetScopeL(MNSmldmDDFObject::EDynamic);
leaf3.SetDffFormatL(MNSmldmDDFObject::Echr);
leaf3.AddDfTypeMimeTypeL(KNSmldMSctsTextPlain);
}

```

The example DDF is just an example and more complicated than normally needed. The idea of this example is to show how to build the DDF structure and also show that there are no limitations in the amount of levels. There can also be several run-time levels in the DDF.

In the previous example, the `DUMMY` and `EXMPLAD` nodes are set as object groups. That is, the URI start which identifies the adapter is `DUMMY/EXMPLAD`. As an example, the commands to the following URIs are directed to the example adapter:

```

DUMMY/EXMPLAD
DUMMY/EXMPLAD/AnyString
DUMMY/EXMPLAD/AnyString/Leaf1
DUMMY/EXMPLAD/AnyString/Node1
DUMMY/EXMPLAD/AnyString/Node1/AnyString/Leaf1
DUMMY/EXMPLAD/AnyString/Node1/AnyString/Node2
DUMMY/EXMPLAD/AnyString/Node1/AnyString/Node2/AnyString/Node3
DUMMY/EXMPLAD/X/Node1/X/Node2/X/Node3/X
DUMMY/EXMPLAD/X/Node1/Y/Node2/Z/Node3/Q/Leaf3

```

The commands to the following URIs are NOT directed to the example adapter:

```

DUMMY (can be figured out from the DDF)
DUMMY/EXMPLAD/AnyString/NotLeaf1 (against the DDF)
DUMMY/EXMPLAD/AnyString/Node3 (against the DDF)
DUMMY/EXMPLAD/X/Node1/Y/Node2/Z/Leaf1 (against the DDF)

```

So, the commands to URIs that are against the DDF are not directed to the adapter, but the dynamic nodes, whose names are empty strings in the DDF, can be anything.

2.4 AddLeafObjectL

Prototype:

```
void AddLeafObjectL( const TDesC& aURI, const TDesC& aParentLUID,
const TDesC& aObject, const TDesC& aType, const TInt aStatusRef )
```

The function is called when the add command comes from the DM server and the command is directed to a URI which belongs to the current adapter. The basic operation is that the adapter adds the data from `aObject` to a field that is pointed by `aURI`. When the adapter has added the data, the success of the operation is given to the engine by calling the `SetStatusL` function of `MNSm1DmCallback`. The callback pointer was a function argument of `NewL`.

The `aParentLUID` is the Local Unified Identifier (LUID) which is mapped to `aURI`. The mapping is done by the adapter, and it can be done via the `MNSm1DmCallback` interface by calling the `SetMappingL` function. When the adapter has mapped a LUID to a URI, the same LUID will come in an `aLUID` argument in all the commands. Also the commands to any of the children of the mapped node will get the LUID in the argument.

If the adapter wants to handle the LUID mapping itself (see Section 2.10, "LuidMappingInAdapterL"), this argument is not needed.

The mime type of data comes in `aType`. Basically this should be the same as told in the DDF structure, but anyhow this comes directly from the DM server.

The status code is given in the callback interface, because in some cases the status cannot be given just when the adapter gets the data, that is, the callback interface makes it possible to buffer the commands. If the adapter wants to buffer the command, `aStatusRef` should be remembered, and `SetStatusL` should be called by using the same status reference.

The following example expects that there is a parent node, which is mapped to the engine. Mapping is shown in 2.9, "AddNodeObjectL."

Example code:

```
void CNSm1DmExmplAdapter::AddLeafObjectL(const TDesC& aURI, const
TDesC& aParentLUID, const TDesC& aObject, const TDesC& /*aType*/, const
TInt aStatusRef)
{
    CNSm1DmAdapter::TError status = CNSm1DmAdapter::ENotFound;

    if (FindRtNode(aParentLUID)) //checks if the parent node is found
    {
        if (LastURISeg((aURI)).Compare(_L("Leaf1"))==0)
        {
            if (LeafHasData()) //checks if the leaf already has the data
            {
                status = CNSm1DmAdapter::EAlreadyExists;
            }
            else
            {
                AddDataToLeaf(aObject);
                status = CNSm1DmAdapter::EOk;
            }
        }
    }

    iCallback->SetStatusL(aStatusRef, status );
}
```

```

}

```

2.5 UpdateLeafObjectL

Prototype:

```

void UpdateLeafObjectL( const TDesC& aURI, const TDesC& aLUID, const
TDesC8& aObject, const TDesC& aType, const TInt aStatusRef )

```

The command operates basically as `AddLeafObjectL`, but this is called when the replace command is received from the DM server. Normally the “add” is made if the leaf does not already exist, but this decision has to be made case by case. If the adapter does the add, the only difference is that the update does not return the `EAlreadyExist` status in any case. Moreover, in some cases it might be impossible to give the `EAlreadyExist` status in the add command either.

Example code:

```

void CNSmldmExmplAdapter::UpdateLeafObjectL(const TDesC& aURI,const
TDesC& aLUID,const TDesC8& aObject,const TDesC& /*aType*/,const TInt
aStatusRef)
{
    if (FindRtnode(aLUID)) //checks if the mapping is found
    {
        if (LastURISeg((aURI)).Compare(_L("Leaf1"))==0)
        {
            if (LeafHasData()) //checks if the leaf already has the data
            {
                UpdateData(aObject);
                status = CNSmldmAdapter::EOK;
            }
            else
            {
                AddDataToLeaf(aObject);
                status = CNSmldmAdapter::EOK;
            }
        }
    }

    iCallback->SetStatusL(aStatusRef, status );
}

```

2.6 DeleteObjectL

Prototype:

```

void DeleteObjectL(const TDesC& aURI,const TDesC& aLUID,const TInt
aStatusRef);

```

The command is executed when the delete command is received from the DM server. The function call is the same for both interior and leaf nodes. The adapter should delete the object which is pointed by aURI. In some cases, it does not make sense to allow deletes to leaf objects, so it can be prohibited in the DDF structure. In case of a successfully made delete, the engine removes the corresponding mapping(s) from the database, that is, the following commands to the same URI (or any of its children) will not get the LUID in commands.

The following example deletes an existing interior node.

Example code:

```

void CNSmldmExmplAdapter::DeleteObjectL( const TDesC& aURI, const
TDesC& aLUID, const TInt aStatusRef )

```

```

{
CNSmldmAdapter::TError status = CNSmldmAdapter::ENotFound;

if (FindRtNode(aLUID)) //checks if the mapping is found
{
DeleteRtNode(aLUID); //deletes run-time node
status = CNSmldmAdapter::EOK;
}
iCallback->SetStatusL(aStatusRef, status);
}

```

2.7 FetchLeafObjectL

Prototype:

```

void FetchLeafObjectL( const TDesC& aURI, const TDesC& aLUID, const
TDesC& aType, const TInt aResultsRef, const TInt aStatusRef );

```

The command is executed when the get command to a leaf object is received from the DM server. The adapter should get the data which is pointed by aURI, and give the result to the engine. The result is given to the engine in a similar way as the status to every command, that is, via the MNSmldmCallback interface, but in this case SetResultsL should be called in addition to SetStatusL. SetResultsL should not be called when the command execution is not successful, so if the returned status is something else than EOK, SetResultsL should not be called. It does not matter which is called first, SetStatusL or SetResultsL. The command buffering can be made as with other commands. The command buffering was explained in Section 2.4, "AddLeafObjectL." In this case, aResultsRef should also be remembered.

Example code:

```

void CNSmldmExmplAdapter::FetchLeafObjectL( const TDesC& aURI, const
TDesC& aLUID, const TDesC& aType, const TInt aResultsRef, const TInt
aStatusRef )
{
DBG FILE("CNSmldmExmplAdapter::FetchLeafObjectL(): begin");
CNSmldmAdapter::TError status = CNSmldmAdapter::ENotFound;

CBufBase *object = CBufFlat::NewL(16);
CleanupStack::PushL(object);

if (FindRtNode(aLUID)) //checks if the mapping is found
{
if (LastURISeg((aURI)).Compare(_L("Leaf1"))==0)
{
object->InsertL(0,LeafData());
status = CNSmldmAdapter::EOK;
}
}

iCallback->SetStatusL(aStatusRef, status );
if(status==CNSmldmAdapter::EOK)
{
iCallback->SetResultsL(aResultsRef,*object,KTextPlain);
}
CleanupStack::PopAndDestroy(); //object
}

```

2.8 ChildURLListL

Prototype:

```
void ChildURIListL( const TDesC& aURI, const TDesC& aLUID, const
CArrayFix<TNSmlDmMappingInfo>& aPreviousURISegmentList, const TInt
aResultsRef, const TInt aStatusRef );
```

The command is executed when the get command to an interior node object is received from the DM server. The adapter should get the list of children of the node pointed by aURI, and give the result to the engine in CBufBase, where the child nodes are separated with "/". The result and status are given in a similar way as with FetchLeafObjectL.

The function also gets the list of children from the engine if there are mappings in the current level. The adapter may add the missing mappings, in addition to making the results list, so the engine will be up to date.

Example code:

```
void CNSmlDmExmplAdapter::ChildURIListL( const TDesC& aURI, const
TDesC& aLUID, const CArrayFix<TNSmlDmMappingInfo>&
aPreviousURISegmentList, const TInt aResultsRef, const TInt
aStatusRef )
{
    CNSmlDmAdapter::TError status = CNSmlDmAdapter::EOk;
    CBufBase *currentList = CBufFlat::NewL(32);
    CleanupStack::PushL(currentList);

    CRtNode *rt = GetFirstNode();
    while(rt)
    {
        currentList->InsertL(0,rt->Name());
        rt = rt->iNext;
        if(rt)
        {
            currentList->InsertL(0,_L8("/"));
        }
        //check if not found from previous list
        if(!SearchPrevList(aPreviousURISegmentList)
        //creates new URI from aURI and rt->Name
        HBufC* mapUri = AddNameToUriLC(aURI,rt->Name())

        //sets the missing mapping
        iCallback->SetMappingL(mapURI,rt->Luid());
        CleanupStack::PopAndDestroy(); //mapURI
    }
    iCallback->SetStatusL(aStatusRef, status );
    if(status==CNSmlDmAdapter::EOk)
    {
        iCallback->SetResultsL(aResultsRef,*currentList,KNullDesC);
    }
    CleanupStack::PopAndDestroy(); // currentlist
}
```

2.9 AddNodeObjectL

Prototype:

```
void AddNodeObjectL(const TDesC& aURI,const TDesC& aParentLUID,const
TInt aStatusRef);
```

The function is called when the add command to an interior node is received from the DM server. The adapter should add the new interior node and give the status as in previous operations. In addition to creating a new node and giving the status back to the engine, the new node may also be mapped to the engine. For example, in case of an e-mail settings adapter, the new e-mail account is created in this function and in creation the uid of the newly created account is received from the messaging API. The uid is

mapped to a URI by calling `SetMappingL` of the `MNSmldmCallback` callback interface. When the mapping is done, the following commands to this node or any of its children will get the uid in command and the alignment is easier.

Example code:

```
void CNSmldmExmplAdapter::AddNodeObjectL( const TDesC& aURI, const
TDesC& aParentLUID, const TInt aStatusRef )
{
    CNSmldmAdapter::TError status;

    //check if node already exists
    if(NodeExists(aURI,aParentLUID))
    {
        status = CNSmldmAdapter::EAlreadyExists;
    }
    else
    {
        //creates new node and returns uid
        TInt luid = CreateNewAccountL();

        //checks if creation was successful
        if(luid)
        {
            CNSmldmAdapter::TError status = CNSmldmAdapter::EOK;
            HBufC *luidBuf = IntToDesLC(luid);
            iCallback->SetMappingL(aURI,*luidBuf);
            CleanupStack::PopAndDestroy(); //luidBuf
        }
        else
        {
            CNSmldmAdapter::TError status = CNSmldmAdapter::EError;
        }
    }
    iCallback->SetStatusL(aStatusRef, status);
}
```

2.10 LuidMappingInAdapterL

Prototype:

```
TBool LuidMappingInAdapter()
```

The function is needed for checking if the adapter wants to do the LUID mapping itself or not. If the adapter returns `ETrue` in this function, the mappings are not looked from the engines database, that is, the adapter will not get the LUIDs in function calls.

The function has the default implementation, which returns `EFalse`.

Example code:

```
TBool CNSmldmExmplAdapter::LuidMappingInAdapter()
{
    return EFalse;
}
```

2.11 AdapterCanChangeACL

Prototype:

```
TBool AdapterCanChangeACL()
```

The function is called when the replace to the Access Control List (acl) property of the node, which belongs to the adapter, is received from the DM server. If the adapter wants

to edit the acls, it should return `ETrue` to this, and then the engine will call `UpdateACLPropertyL` (see 2.12, “UpdateACLPropertyL”) when needed.

The function has the default implementation, which returns `EFalse`.

Example code:

```
TBool CNSmldmExmplAdapter::AdapterCanChangeACL()
{
    return EFalse;
}
```

2.12 UpdateACLPropertyL

Prototype:

```
void UpdateACLPropertyL(const TDesC& aURI, const TDesC& aLUID, const
TDesC8& aACL, const TInt aStatusRef);
```

The function is called when the replace to the acl property of the node and the adapter has returned `ETrue` to `AdapterCanChangeACL` (see Section 2.11, “AdapterCanChangeACL”). The function gets the acl in the same format as it comes from the DM server; the format is specified in *SyncML Device Management Tree and Description* [3]. The acls are always set to the engine database, that is, the adapter has to set the edited acl by calling `SetACL` of the `MNSmldmCallback` callback interface. The function returns `CNSmldmAdapter::TError`, which can be given to the engine in the same way as the status to other commands too. The acls must be edited very carefully. By doing mistakes here, the adapter may, for example, remove all the access rights by mistake.

The function has an empty default implementation, that is, there is no need to implement this if the adapter does not want to edit acls.

Example code:

```
void CNSmldmExmplAdapter::UpdateACLPropertyL( const TDesC& aURI,
const TDesC& aLUID, const TDesC8& aACL, const TInt aStatusRef )
{
    HBufC8 *acl = EditAclLC(aACL);
    CNSmldmAdapter::TError status = iCallback->SetACL(aURI, aLUID, *acl);
    iCallback->SetStatusL(aStatusRef, status);
    CleanupStack::PopAndDestroy(); //acl
}
```

2.13 EndMessageL

Prototype:

```
void EndMessageL()
```

The function is called at the end of every packet. The meaning of this function is to note that there are no more commands in the packet and the adapter must give the status and results to all the buffered commands. The function has an empty implementation as default, and so the adapter does not need to implement this if it does not buffer the commands.

Example code:

```
void CNSmldmExmplAdapter::EndMessageL()
{
    //gets the first buffered command
    TCommand *command = FirstBufferedCommand();
}
```

```
//executes all the commands and saves the status and results to
//command
ExecuteCommandsL(command);

//loops all the commands and gives the status and results to engine
//NOTE! the iStatusRef and iResultsRef must be stored when the
//actual request is made by engine
while(command)
{
    iCallback->SetStatusL(command->iStatusRef,command->iStatus);
    if(command->HasResults() &&command->iStatus==CNSmlDmAdapter::EOk)
    {
        iCallback->SetResultsL(command->iResultsRef,*command-
>iObject,*command->iMimeType);
    }
    command=command->iNext;
}
//delete command buffer
DeleteBufferedCommands();
}
```

3. Terms and abbreviations

Term or abbreviation	Meaning
ACL	Access Control List
API	Application Programming Interface
DDF	Device Description Framework
DM	Device management
LUID	Local Unified Identifier
SDK	Software Development Kit
SyncML	Synchronization Mark-up Language
URI	Unified Resource Identifier

4. References

- [1] API Reference, Series 60 2nd Edition SDKs for Symbian OS
- [2] ECOM Public API v1.02.chm, Symbian documentation
- [3] SyncML Device Management Tree and Description, ver 1.1.1, <http://www.openmobilealliance.org/syncml/>
- [4] [S60 3rd Edition: Device Management Plug-in](#), available at <http://www.forum.nokia.com>

5. Evaluate this resource

Please spare a moment to help us improve documentation quality and recognize the resources you find most valuable, by [rating this resource](#).