

Nokia 7710: Messaging APIs

Version 1.0; May 24, 2005

Nokia 7710

NOKIA

Copyright © 2005 Nokia Corporation. All rights reserved.

Nokia, Nokia Connecting People and Nokia 7710 are registered trademarks of Nokia Corporation. Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. 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	Introducing Messaging	6
2.1	Common Messaging Concepts.....	6
2.2	The Messaging Server	7
2.2.1	The role of the local service.....	8
2.2.2	Messaging operations	8
3	Simple Send APIs	9
3.1	Simple Sending	9
3.2	Simple Messaging UI	10
3.3	Using the Send As Dialog	12
4	Basic Messaging APIs and MTMs	13
4.1	Messaging Server Session.....	13
4.1.1	Manipulating message items.....	14
4.1.2	Creating a new entry	14
4.2	Using MTM APIs	17
4.3	MTM Registries.....	17
4.4	Base Client MTM API.....	18
4.5	Using the MTM UI.....	20
5	Conclusion	26
6	Terms and Abbreviations	27
7	References	28
8	Evaluate This Document	29

Change History

May 24, 2005	V1.0	Initial document release

1 Introduction

Multimedia Messaging Service (MMS) provides a mechanism for mobile device users to create, send, and receive rich messages that include text, images, and sounds. It also provides developers with numerous possibilities for adding functionality to applications and offering users exciting communication options. Using MMS requires an understanding of both the Symbian OS messaging framework and the structure and content of MMS.

This is the first in a series of three documents that provide a detailed description of the messaging capabilities of the Nokia 7710 multimedia smartphone and how these capabilities can be used to create, send, and receive MMS messages programmatically.

The three documents are:

- ◆ *Nokia 7710: Messaging APIs* — Introduces the messaging subsystem provided by Symbian OS and describes the fundamental Symbian OS messaging APIs and Message Type Modules (MTMs), as well as how to use those basic messaging APIs.
- ◆ *Nokia 7710: Using The Messaging APIs For MMS* — Describes how to use the Symbian OS messaging APIs to create, send, and receive MMS messages.
- ◆ *Nokia 7710: Creating MMS Content* — Describes how to programmatically create the content of an MMS message and provides an introduction to Scalable Vector Graphics.

While these documents are based on the messaging and MMS capabilities of the Nokia 7710 smartphone, much of the content is applicable to both Series 60 and Series 80 Platforms and will be useful to developers wishing to create messaging or MMS applications for those platforms as well.

2 Introducing Messaging

This document will familiarize readers with the concepts behind the Nokia 7710 and Symbian OS messaging subsystem. After reading it, they will have an understanding of the terminology associated with messaging and will be able to use Nokia 7710's simple APIs to send messages.

The Nokia 7710 smartphone messaging subsystem is based on and extends the standard Symbian OS messaging framework. This chapter introduces the concepts and APIs that are common for all messaging-related tasks, and then examines the message type-specific terminology in detail.

2.1 Common Messaging Concepts

At the very core of the messaging framework is the *Messaging Server*. The Messaging Server is a process (or a thread in a single-process environment such as the emulator) that is responsible for storing and manipulating *message entries* at the lowest level. Messaging entries are the basic units upon which the messaging applications operate; these entries represent messages but also other messaging information such as account settings or folders. Messaging entries may contain stores, known as *message stores*, which are storage structures for their associated data.

Working with the server are the *Message Type Modules (MTMs)*. Each MTM is a software plug-in that is responsible for handling a single message type — Short Message Service (SMS) or MMS, for example. Typically, each MTM has its own special message entry called a service entry, which is used for storing the service settings. Some MTMs have more than one service entry, for example, there will be several POP3 MTM service entries if the user has to configure more than one POP mailbox. MTMs are further divided into *server*, *client*, *UI*, and *UI data* MTMs, which are separate components (such as the SMS Client MTM component), but as a group are sometimes referred to as one MTM (such as the SMS MTM). This dual meaning for MTM can be confusing, and unfortunately both meanings are frequently found in documentation.

The server MTM is responsible for performing the send and receive operations for the message type. It is usually the server MTM that is responsible for encoding and decoding the messages or protocol data units (PDUs) on their respective protocol level. For example, the MMS server MTM sends MMS PDUs over the WAP and HTTP networks. The name "server MTM" comes from the fact that it runs in the Messaging Server process or thread. Only developers who plan to implement a completely new message type will need to understand the server MTM API or the particulars of its implementation, and because the creation of new message types is rare, these aspects of the MTM will not be discussed further.

The client MTM is an application's main contact point for accessing the MTM-specific functionality. As the name implies, the client MTM is used by the Messaging client application to invoke the services of the server MTM. By using the client MTM APIs, the application can, for example, configure the attachments or delivery options for an MMS message or retrieve and set the details of the MMS account. It is worth noting that the client MTM APIs only need to be used if an application has to use services specific to an MTM. For simple operations there are generic messaging APIs such as *Send As* and *Send UI*, which may be sufficient for an application's needs (however they are simply wrappers around the MTM functions and ultimately by using them the application is indirectly using the client MTM services).

The UI MTM provides access to a higher level of functionality. Rather than manipulate the message according to an application's commands, the UI MTM allows an application to invoke the standard platform user interface components for handling the message or service entry. In other words, the UI MTM API can be used to open the default editor for a message type, or to open the account settings dialog, and then allow the application user to determine the messaging actions to be undertaken.

The UI data MTM is complementary to the UI MTM and acts as a lightweight helper component that stores static information about the UI MTM. The UI data MTM is used to query the MTM capabilities and

the set of supported functions without instantiating the UI or Client MTM, which are heavyweight components.

2.2 The Messaging Server

The Messaging Server is the core of the Nokia 7710 smartphone messaging capability. Most of its tasks are performed on *message entries*. An entry is usually a message, but it may also be a folder, a service, or an attachment. As shown in Figure 1, the message entries form a hierarchical structure, with a special entry known as the *root entry* at the top of the tree. Under the root entry, there are message type-specific service entries. In the Nokia 7710 smartphone, these entries include MMS service, SMS service, Push service, and zero or more SMTP, POP3, and IMAP4 services. There is also the “archive” pseudo-service, which is not designed to be used by client applications. Moreover, there is a special system service known as the local service, which is not specific to any message type. The local service represents the local storage for messages. It consists of Inbox, Outbox, Drafts, and Sent folders, as well as any custom folders the user may have created. There is also the Bluetooth MTM, which is an exception in that it does not have a corresponding service entry.

The entries under a service are service-specific. Server MTMs may use them for storing temporary data, or they may not have any structure there at all — in that case, the service entry merely acts as the gateway for messages to the outside world. E-mail MTMs store the received messages under the service rather than in the Inbox folder by default (this behavior may be changed by the user).

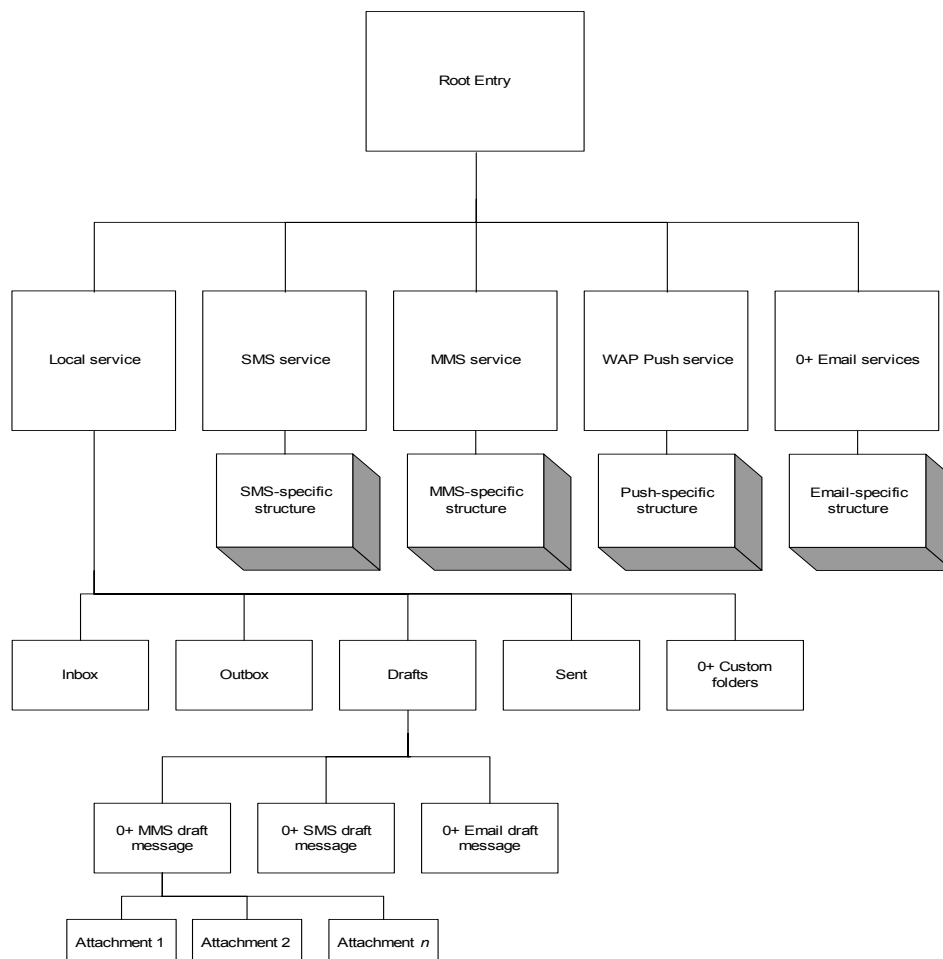


Figure 1: The hierarchy of messaging entries with an example branch expanded

2.2.1 The role of the local service

When a Nokia 7710 smartphone receives a message it is usually delivered to the local service Inbox; e-mail MTMs are the exception, as previously mentioned. Similarly, any message that is sent from a Nokia 7710 smartphone, be it MMS, SMS, or Bluetooth object transfer, will start its outbound journey from the local service Outbox.

While it is possible to create folders underneath the Inbox, Outbox, Drafts, and Sent folders, it is not normally permitted by the Messaging Center application because they serve no particular purpose. Therefore, the Inbox, Outbox, Sent, Drafts, and custom folders contain messages only. There might be attachment entries under messages, if the given message type supports them.

Message entries are, to some extent, an abstract concept because they are the units upon which copy, move, or delete operations, which are abstract themselves, are performed. In other words, message entries can be copied, moved, or deleted using a common API, but the semantics of these operations differ, depending on the service under which or to which the operations are performed, the type of entry, etc. For example, if an entry is copied between two folders under the local service, it will simply create a binary copy of the entry in a new place. However, if a message entry is copied to the SMS service entry, it will result in the message being sent.

A more concrete concept is the *message store*. A store is an object associated with an entry containing entry-specific data. For instance, the store associated with an SMS message entry will contain the body text of the message as well as the recipient or sender information, and the store associated with the MMS service entry will contain the MMS configuration settings. While the data structure in the store is MTM-specific, some MTMs (for example, SMS, e-mail) support a common interface for retrieving the body of a message in rich text format. MMS is an example of an MTM that does not support this type of access for the message body.

Figure 1 shows the message entry hierarchy with one of the branches of the tree expanded. Note that the expanded part appears under the local service. The structure of the tree under the other nodes varies from service to service. Usually any entry under a service represents a resource that is located across the network. For example, if there are message entries under a POP3 service, they represent mail stored on a remote server. When such an entry is copied to the local service, the mail is downloaded from the server to the device. Hence, all entries that are not under the local service are called *remote entries*. Note that a remote entry is usually only a *representation* of a remote resource. The entry itself is stored within the local messaging hierarchy.

2.2.2 Messaging operations

An *operation* is a function performed by messaging, such as creating, deleting, or copying an entry. The semantics of an operation differ depending on the context. Some operations may be performed relatively quickly (for example, a local delete operation), whereas others may involve a lengthy process (for instance, a remote copy operation). The lengthy asynchronous functions have operation objects associated with them. By querying these objects it is possible to check aspects of the operation such as the amount of data transferred.

Synchronous operations can be used within the local service only. With remote entries, an asynchronous version of an operation must always be used.

3 Simple Send APIs

This chapter examines the simple set of Send and Send As APIs that allows developers to add send capabilities to applications without significant programming overhead. Chapter 4 will take a look at the messaging APIs that allow messages to be manipulated in detail — those APIs are complex and often more than is needed for many applications.

3.1 Simple Sending

The class `CSendAs` provides the simplest sending capability, and is another wrapper around the client MTM API. To instantiate an object of this class, it is necessary to implement `MSendAsObserver`. The interface has four methods, but three are to support printing, which is not available in the Nokia 7710 smartphone, and can thus be ignored. The only method that must be implemented is `virtual TBool CapabilityOK(TUId aCapabilty, TInt aResponse)`, which returns a Boolean value and can be used to check the MTM capabilities. A trivial implementation of this function will usually be enough. The `CSendAs` object is instantiated by the simple `NewL()` static method, which requires a reference to an object implementing the `MSendAsObserver` interface.

With the object created, the next step is to set the MTM that will be used to send the message. One way of doing it is by calling the `SetMtmL()` method and passing in the MTM UID. If the UID is not known or the application needs to present the choice of MTMs to the user, a human-readable list of MTM names can be obtained by calling the `AvailableMtms()` method, which returns an array of descriptors. Once the MTM has been selected, the other overload of `void SetMtmL(TInt aMtmIndex)`, which takes an index to the array as the parameter, can be used to set the MTM.

After the message type has been selected, it is necessary to choose the service that will be used for sending. Most MTMs only have one service, so there is no choice to be made, but in some cases, such as e-mail, several accounts could be used. The services available for the MTM can be enumerated using the `AvailableServices()` method, which again returns an array of human-readable strings; then the service can be chosen by calling `SetService()` and passing an index to the array. Having made a choice, it is now possible to create a new empty message by calling `CreateMessageL()`. The new message will be created in the Drafts folder unless the variant of the `CreateMessageL()` method that takes a `TMsvId` parameter is used to force the entry to be created in a specific folder. Once the message has been created, the following functions can be used to create its content:

```
const CDesCArray& RecipientList() const,
void AddRecipientL(const TDesC& aRealAddress, const TDesC& aAlias), void
    AddRecipientL(const TDesC& aRealAddress),
void RemoveRecipient(TInt aIndex),
void SetSubjectL(const TDesC& aSubject),
void SetBodyL(const CRichText& aMessageBody),
TMsvPartList ValidateMessage(),
TInt QueryMessageCapability(TUId aCapabilityId, TBool
    aResponseExpected=EFalse),
void CreateAttachmentL(TMsvId& aAttachmentId, TFileName& aDirectory),
void DeleteAttachmentL(TMsvId aAttachmentId),
void SetAttachmentL(TMsvId aAttachmentId, const TDesC& aAttachmentName).
```

The functions map directly to the base client MTM API methods; for more information on their use please refer to the discussion in Section 4.4, “Base Client MTM API.”

In addition, there is the method `SetBioTypeL()`, which selects the type of BIO messaging (smart messaging) that will be used. This method only makes sense for bearers that support BIO messaging, such as SMS.

Once the message detail has been added, the message should be saved using `SaveMessageL()`, which has both synchronous and asynchronous variants, and has a default Boolean parameter that can be used to change the visibility of the message to the user. Alternatively, message creation can be cancelled with the `AbandonMessage()` method, which performs the cleanup.

3.2 Simple Messaging UI

The simplest way to add messaging functionality to an application is by using the *Send UI* component. This component offers two interrelated services for an application: The first service is the “Send as...” menu cascade, which can easily be plugged into an application and customized to the application's specific needs; the second service is the Send function, which creates an outgoing message (possibly invoking the message editor user interface) with the body, attachments, and recipients, all with a single line of code. The compromise made here is that the easy to use simple messaging APIs don't provide fine control over the message.

The main class used to implement Send UI functionality is `CSendAppUi`. An instance of this class is created as a member variable of the application's UI component (the class is derived from `CEikAppUi`). The class should be instantiated early, rather than when the application is about to be sent — ideally at application startup because the initialization of the Send UI object involves complex operations and takes a relatively long time. The static `CSendAppUi* NewL(const TInt aCommandId, CEikHotKeyTable* aHotKeyTable = NULL)` method of the class, which is needed in order to create a new Send UI object, uses the optional `aHotKeyTable` parameter, which is a pointer to, and allows the configuration of, an accelerator key for the send command. The pointer to `CEikHotKeyTable` is easy to obtain in the application UI constructor, but the use of this feature only makes sense for devices that have a keyboard.

The `aCommandId` parameter passed at the construction to the `NewL()` method is the command ID that will be associated with the Send menu cascade and is used as the base for the command ID for each MTM. Because the number of MTMs available for sending cannot be determined in advance, a base ID that does not clash with other commands in the application must be selected.

To display the Send menu item, the application must implement the virtual `void DynInitMenuPaneL(TInt aResourceId, CEikMenuPane* aMenuPane)` method in its UI. The `aResourceId` parameter is the resource ID of the menu that will contain the Send As option; as shown in Figure 2, the call for this pane is handled by invoking `CSendUi::DisplayMenuItemL()` and passing the pane pointer to it, as well as the index number that controls where in the pane the command will appear.

The other parameters to the `DisplayMenuItemL()` function are two `TSendingCapabilities` objects that indicate the available and displayed capabilities (the latter is optional). With `TSendingCapabilities` it is possible to filter the MTMs to limit their availability to the user. This can be done in a number of ways. The `TSendingCapabilities` constructor allows the total (body and attachments) message size to be specified, so only those MTMs that send messages of a certain size are available. Moreover, the application can specify the required body size in a similar way. Finally, bit flags can be used to control other conditions that the MTMs must fulfill, such as support for sending message body, attachments, and BIO messaging. If two `TSendingCapabilities` objects are passed to the `CSendUi::DisplayMenuItemL()` function, the first will control which MTMs are visible in the menu, and the second will dim the message types not available for the particular send action being undertaken.

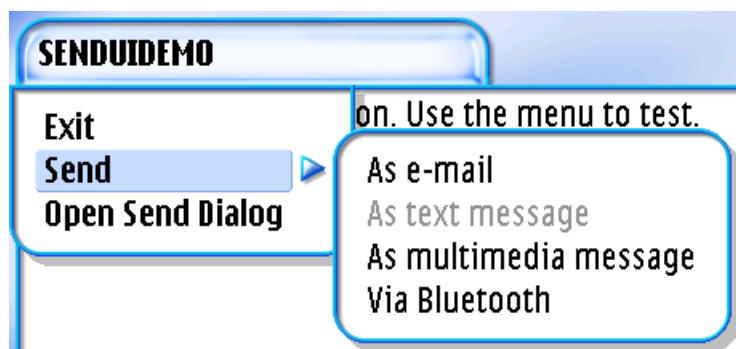


Figure 2: The menu cascade created by the Send UI component

To display the cascade menu, a separate call to `DynInitMenuPaneL()` is required, this time passing the resource ID that corresponds to the cascade, which is `R_SENDUI_MENU` from `SendNorm.rsg`. The pane pointer returned by `DynInitMenuPaneL()` has to be given to virtual void `CSendAppUi::DisplaySendCascadeMenuL(CEikMenuPane& aMenuPane, CArrayFix<TUid>* aMtmsToDim, CArrayFix<TUid>* aMtmsToDrop)`, which will display the cascade menu. The `aMtmsToDim` parameter can be passed to the function — a pointer to a dynamic array of UIDs that allows the choice of MTMs to be further limited, by dimming MTMs defined in the array.

When the user selects an item from the Send menu, the command is handled in the `HandleCommandL()` method of the application UI. If the command ID is within the range specified for the send commands, the command is handled by invoking the appropriate send function. There are two overloads of `CreateAndSendMessageL()`:

```
virtual void CreateAndSendMessageL(const TInt aCommandId, const
CRichText* aBodyText = NULL, MDesC16Array* aAttachments = NULL, const
TUid aBioTypeUid = KNullUid, MDesC16Array* aRealAddresses = NULL,
MDesC16Array* aAliases = NULL),
```

```
virtual void CreateAndSendMessageL(const TUid aMtmUid, const CRichText*
aBodyText = NULL, MDesC16Array* aAttachments = NULL, const TUid
aBioTypeUid = KNullUid, MDesC16Array* aRealAddresses = NULL,
MDesC16Array* aAliases = NULL)
```

One takes an MTM UID and the other takes the command ID. Both do exactly the same thing, and a command ID can be translated to the MTM UID by calling `CSendAppUi::MtmForCommand()`. The function will invoke the default user interface for composing and sending the message, which usually is some kind of message editor (the Bluetooth MTM is an exception in that it has no real editor but only the recipient selection dialog). The other parameters that can be given to the two methods are:

A message body — `aBodyText` in the form of a `CRichText` pointer.

A list of attachments — `aAttachments` pointer to `MDesC16Array` containing the attachment file paths.

A list of addresses — `aRealAddresses` pointer to `MDesC16Array` containing the real addresses (addresses are specific to the message type, so the application should check the message type and format or provide proper addresses).

A list of addressee aliases — `aAliases`, another pointer to `MDesC16Array`.

All these parameters have default values, so it is not necessary to specify any of them to create an empty message. Of course, most applications will specify at least one parameter, since creating an empty message with no addressee is hardly ever useful. A second method, `ExtendedCreateAndSendMessageL()`, permits the definition of the subject as well as CC and BCC recipients for the message. Obviously it will only work for message types that support CC and BCC fields, which is not the case, for example, with SMS.

3.3 Using the Send As Dialog

The alternative implementation is to use the Send As dialog, shown in Figure 3. The dialog is invoked with a call to `TBool SendDialogL(TUid& aMtmUid, TSendingCapabilities aRequiredCapabilities, TSendingCapabilities aCapabilitiesToDisplay = TSendingCapabilities(), TInt aDimmedItemInfoResId=-1, CArrayFix<TUid>* aMtmToDrop = NULL)`, which has an analogous functionality to the cascade. The function returns a Boolean value indicating whether an MTM selection was completed by the user (`ETrue`) or cancelled (`EFalse`). If it was accepted, the function returns the UID of the selected MTM via the first reference parameter.

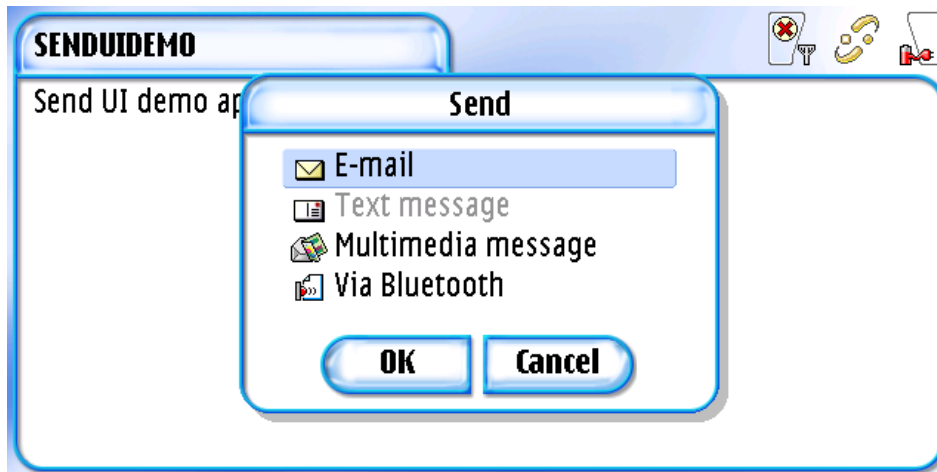


Figure 3: The Send UI dialog offers an alternate implementation

4 Basic Messaging APIs and MTMs

This chapter examines the basic messaging APIs and MTMs that allow messages to be manipulated in detail.

4.1 Messaging Server Session

The previous chapter described the Messaging Server as the core of the messaging subsystem. Not surprisingly then, any application that wants to use messaging service will need to establish a *session with the Messaging Server*. If an application uses the Send UI API, the session does not have to be established explicitly because it is done by the API.

Most of the messaging-related classes can be distinguished by the use of Msv infix in the class name.

Unlike many other services in Symbian OS that are accessed by using an R-class client side resource handle, connecting to the Messaging Server involves a different approach. The R-class, `RMsvServerSession`, does exist, but it is not normally used. It is a protected member of `CMsvSession`. In order to connect and use the server, an implementation of the `MMsvSessionObserver` interface is needed. In practice, implementing this interface means defining a single method, `HandleSessionEventL()`, which is invoked whenever a Messaging Server session event occurs.

Typically, an application is only interested in handling the `EMsvServerReady` event, which is fired after the session is created to indicate that it is now ready to use. An attempt to use the session before it is ready will fail. However, an application should be able to handle `EMsvServerFailedToStart` and `EMsvCloseSession` events. The former is unlikely to occur because in most cases the Messaging Server will already be running, as it is started when a device is turned on. Still, it is possible to stop the server by using the `CMsvSession::CloseMessageServer()` method. If that is done and a subsequent attempt is made to restart the server but, for example, a low memory condition occurs, `EMsvServerFailedToStart` may be returned rather than `EMsvServerReady`.

The second event that an application should be always prepared to handle — `EMsvCloseSession` — fires when the Messaging Server is shutting down, for example, because the device is shutting down or rebooting, or another application is requesting the server to close. In such cases, the application should *immediately* terminate any outstanding operations with the server, release all messaging objects, and then close the session. It is not necessary for an application to handle any other event that can be reported to its implementation of `MMsvSessionObserver`, but, of course, there may be times when it is prudent to do so.

Unless a test application is being written or code experimentation is taking place, attempts to shut down the message server should *not* be made. Doing so will disrupt the operation of other applications.

Once the `MMsvSessionObserver` interface has been implemented, the application is ready to instantiate the session. In the `CMsvSession` class, two static functions can be used to create a new session:

```
OpenSyncL(MMsvSessionObserver& aObserver),
OpenAsyncL(MMsvSessionObserver& aObserver),
```

Both functions return a new `CMsvSession` object. `OpenSyncL()` returns an object that is immediately ready for use (unless something goes wrong), while `OpenAsyncL()` returns an object that is not to be used until the `EMsvServerReady` event fires. If active objects and other Symbian OS

asynchronous services are used properly in the application, then `OpenAsyncL()` is the obvious function to use, since `OpenSyncL()` involves unnecessary blocking, which prevents the application from performing other tasks while the session is starting. Note that only one session with the Messaging Server is allowed per execution thread.

Now the session is connected, and the application can start using the Messaging Server's services.

4.1.1 Manipulating message items

Once a session has been established with the Message Server, the application can begin to manipulate message items.

An entry is represented by the `CMsvEntry` class instance. There are many ways of obtaining an entry from Messaging — one is simply to call the static `NewL()` method of `CMsvEntry`; another one is to use the `GetEntryL()` method of `CMsvSession`. There are also a number of other methods of instantiating new `CMsvEntry` objects by performing some operations. Note that all these methods create a new `CMsvEntry` object, not a new messaging entry. This is an important distinction because a new `CMsvEntry` object in this case is only a new representation of an *existing* entry. In fact, both methods of obtaining a new `CMsvEntry` require passing an identifier for an existing entry as a parameter. The identifier is of the `TMsvId` type and is used by messaging to uniquely identify entries. When a new entry is created, messaging takes care of assigning it a unique ID.

Another class, `CMsvServerEntry`, embodies essentially the same concept, but is intended for use by server-side MTMs only. Developers planning to write such a component will have to use the `CMsvServerEntry` class. In contrast to client applications, which only instantiate `CMsvEntry` class objects when needed, a server MTM always has an instance of the `CMsvServerEntry` class. However, when a server entry object is set to point to an entry, that entry is locked. Server MTMs should park their `CMsvServerEntry` on a special NULL entry (`KMsvNullIndexEntryId`) whenever they are not actively using it.

4.1.2 Creating a new entry

In order to create a new entry, the application must instantiate a `CMsvEntry` object corresponding to an existing entry and then invoke one of its `CreateL()` overloaded methods. This will create a new entry under the existing one. Using the `CreateL()` methods, as well as many others, requires a `TMsvEntry` type parameter. `TMsvEntry` is a class representing an index entry — a “summary” of an entry stored in the index file of the Messaging Server. An index entry contains many important pieces of information describing the entry, such as the type of entry, UID of its associated MTM, and ID of the service to which the entry belongs.

Two important members of `TMsvEntry` are the `iDescription` and `iDetails` descriptors, which the Messaging Center application uses when showing the list of messages to the user. It is possible to save any arbitrary text in these structures, but typically `iDetails` contains the sender/recipient while `iDescription` contains the subject information, so it is advisable to adopt this pattern to maintain consistency with the Nokia 7710 smartphone messaging style. The following code sample shows how to update the description of an entry:

```
CMsvEntry* entry = iSession.GetEntryL(KMyEntryId);
CleanupStack::PushL(entry);
TMsvEntry indexEntry (entry->Entry());
indexEntry.iDescription.Set(_L("My new description"));
indexEntry.iDate.HomeTime(); // Update the modification date
entry->ChangeL(indexEntry); // Save the changes
CleanupStack::PopAndDestroy(entry);
```

Creating a new entry requires access to an existing message system entry, such as a root or folder (normally a folder), under which the new entry will be placed. Instantiating a `CMsvEntry` object for

an existing entry requires a `TMsvId` type parameter. To find an existing message identifier, the messaging tree must be navigated until the required entry is found. An identifier is required to initiate the search; it starts with a set of predefined IDs provided by `MsvIDs.h` that are used to instantiate the first `CMsvEntry` from which the search commences.

If the search strategy adopted is to traverse the entire tree, `KMsvRootIndexEntryId` will identify the messaging root folder. The “whole tree” method is necessary when a suitable entry’s location is unknown; however, if an entry is known to exist in a specific folder then a specific ID can be used. Thus, to start a search in the Sent folder, the `KMsvSentEntryId` will be used to instantiate an appropriate `CMsvEntry` object. Once a branch of the tree has been defined, the search can traverse down the tree using one of the child entry enumeration methods: `ChildrenL()`, `ChildrenWithServiceL()`, `ChildrenWithMtmL()`, or `ChildrenWithTypeL()`. Each of these functions returns an object of `CMsvEntrySelection`. This class is simply an array of `TMsvIds`. For example, the following will create an array of all MMS message entries in the Sent folder:

```
CMsvEntry* entry = session->GetEntryL(KMsvSentEntryId);
CleanupStack::PushL(entry);
CMsvEntrySelection* selection = entry-
>ChildrenWithMtmL(KUIdMsgTypeMultimedia);
CleanupStack::PushL(selection);
```

Invoking `CMsvEntry::CreateEntryL()` will create a new but empty entry under the current one. For simple entries such as folders, this will be all that is needed; message entries, however, will require message type-specific structures to be present in the store. In addition to inserting the body text, MTMs also require their specific streams to be present in the store to be able to send the message. Such streams typically contain recipient information, message configuration settings, and other details. Creating such streams inside every entry created would be extremely difficult if not impossible without using the client MTM APIs.

There will be cases where it is necessary to manipulate the contents of a message store, for instance if there is a requirement for the application to associate additional (nonsendable, persistent) data with every message. This involves the usage of the `CMsvStore` class, which is instantiated from a `CMsvEntry` class object by calling its `ReadStoreL()` method. Again, instantiating a new `CMsvStore` in this way does not create a new store, but simply creates a new object that represents an existing store.

`ReadStoreL()` provides read-only access to the store. To be able to write to the store, `WriteStoreL()` of `CMsvEntry` must be invoked. Doing so also returns an object that points to an existing store, but unlike `ReadStoreL()`, `WriteStoreL()` will create one if needed. Because `ReadStoreL()` does not create a store, the existence of a store should be confirmed by calling `CMsvEntry::HasStoreL()` before attempting to read from the store, otherwise the reading function may leave with the error code `KErrNotFound`.

The message store is a kind of Symbian OS *dictionary store* — a store where all streams are accessed through a UID, rather than directly by stream ID — though it derives directly from `CBase`. Message entries can store a number of streams; each of them is identified by a UID.

There is one special stream in the store: the *body text*. This name may cause confusion — the body of a message could quite possibly be a binary object stored inside a rich text object, as is often the case with certain BIO message types such as ring tones or operator logos. Some other message types such as MMS have message headers and attachments but no body at all. The body text is saved and loaded from the store using the `StoreBodyTextL()` and `RestoreBodyTextL()` methods, respectively. Again, before attempting to read the body it is wise to ensure that it really does exist by calling `CMsvStore::HasBodyTextL()`.

To check for other streams, use the `IsPresentL()` method, which takes a `TUId` type parameter. There is no method in `CMsvStore` that allows a new stream to be created or an existing one to be accessed. There are two classes, `RMsvReadStream` and `RMsvWriteStream`, whose construction

methods take the `CMsvStore` reference parameter and behave like the standard Symbian OS read and write streams. If there is a requirement to store extra information in a message entry, a new stream with its own UID can be created and the data serialized there. For instance, suppose a developer is writing a message editor application that saves data per-message editor configuration. Such data can be associated with a message by saving it in the same store, but in a different stream. This can be done even if the store already contains other streams specific to the message type. Typically, MTMs will ignore any streams that they do not recognize.

The code below shows how to create a new draft entry and save body text plus additional information:

```
/* It is assumed that a pointer to the session stored in iSession
already exists, the UID of the applications private stream is
KMyStreamUid. */

CMsvEntry* entry = iSession->GetEntryL(KMsvDraftEntryId);
CleanupStack::PushL(entry);
TMsvEntry indexEntry;
indexEntry.iType = KUidMsvMessageEntry; // It's a message
indexEntry.iMtm = KUidWhateverMtm;
indexEntry.iDescription.Set(_L("This is my new message."));
indexEntry.iDetails.Set(_L("It is to my friend."));
entry->CreateL(indexEntry);
iSession->CleanupEntryPushL(indexEntry.Id());

// Now switch the entry object to point to the new one
entry->SetEntryL(indexEntry.Id());
CMsvStore* store = entry->EditStoreL();
CleanupStack::PushL(store);
RMsvWriteStream stream;
stream.AssignLC(*store, KMyStreamUid);
stream.WriteL(_L("This is my data saved in the store."));
stream.CommitL();
CleanupStack::PopAndDestroy(); // close stream
store->CommitL();
// Now save some body text
CParaFormatLayer* paraLayer = CEikonEnv::NewDefaultParaFormatLayerL();
CleanupStack::PushL(paraLayer);
CCharFormatLayer* charLayer = CEikonEnv::NewDefaultCharFormatLayerL();
CleanupStack::PushL(charLayer);
CRichText* richText = CRichText::NewL(paraLayer, charLayer);
CleanupStack::PushL(richText);
richText->InsertL(0, _L("This is body text."));
store->StoreBodyTextL(*richText);
store->CommitL();
CleanupStack::PopAndDestroy(4, store); // store,
// paraLayer, charLayer, richText
iSession->CleanupEntryPop(); // remove cleanup entry
CleanupStack::PopAndDestroy(entry);
```

Note the usage of `CMsvSession::CleanupEntryPushL()` and `CleanupEntryPop()` in the code. The functions are almost similar to the cleanup functions of the `CleanupStack` class, which will be familiar from general coding. The important difference is that instead of putting a memory cleanup item or pointer on the stack, or removing it, the methods of `CMsvSession` place an entry deletion item on the stack. If any function between the `CleanupEntryPushL()` and `CleanupEntryPop()` invocations leaves, the entry cleanup would be performed. In other words, if something leaves before the cleanup item is popped, the entry will be deleted. This is a very convenient way to ensure that an incomplete entry is not left in the server when, for example, an out-of-memory condition occurs. Of course, there may be cases when this is not desired and an incomplete entry is preferable to no entry at all.

4.2 Using MTM APIs

Working with message entries and stores is not practical for most applications, because of the complexity involved and the need to know a great deal about the inner workings of the message type in question. MTM APIs provide a higher level of abstraction and are generally more straightforward to work with.

There are four types of MTMs: server, client, UI, and UI data. They can be directly mapped to four Symbian OS classes: `CBaseServerMtm`, `CBaseMtm`, `CBaseMtmUi`, and `CBaseMtmUiData`, respectively. The name `CBaseMtm` may suggest that it is the base class for the other MTM classes, but it is not. There is no common base class for all the MTM classes apart from the obvious `CBase`, from which all C classes in Symbian OS derive.

All of the above-mentioned classes are abstract. Each message type implements its own concrete classes that derive from them. For instance, MMS message type implementation consists of MMS server MTM, MMS client MTM, MMS UI MTM, and MMS UI data MTM concrete classes.

4.3 MTM Registries

To instantiate an object of an MTM class is not as simple as using a C++ constructor or Symbian OS standard static `NewL()` method. There is no `NewL()` method in any of the MTM classes and all constructors are protected. The correct way of obtaining an object of any of the client, UI, or UI data MTM classes is through its respective *registry*. Before a concrete MTM object can be retrieved it is necessary to instantiate a registry object matching the MTM to be retrieved. The three classes are:

- `CClientMtmRegistry` — For retrieving `CBaseMtm`-derived objects.
- `CMtmUiRegistry` — For retrieving `CBaseMtmUi`-derived objects.
- `CMtmUiDataRegistry` — For retrieving `CBaseMtmUiData`-derived objects.

Each of these classes has a simple `NewL()` method, which takes a `CMsvSession` reference parameter and an optional `TTimeIntervalMicroSeconds32` parameter. Therefore, a *ready* session with the Messaging Server is required before a registry object can be instantiated. The optional parameter specifies the timeout after which an MTM object will be unloaded once it has been released. In other words, when an MTM is obtained from a registry and subsequently released, it is not unloaded immediately but only after the specified timeout. This ensures a timely response if another instance of the MTM is needed in quick succession.

Note: There is also a fourth registry class, `CServerMtmDllRegistry`, from which an instance of the server MTM could be obtained. This, however, is not intended for use by applications. Server MTMs should never use directly from a client application, but only via a client MTM.

Each of the registries has a method for retrieving an MTM object:

- `CClientMtmRegistry::NewMtmL(TUid aMtmTypeUid)` — Takes a message type UID as a parameter and returns a pointer to an object of `CBaseMtm`-derived class (i.e., a client MTM).
- `CMtmUiRegistry::NewMtmUiL(CBaseMtm& aMtm)` — Takes a reference to the corresponding client MTM as a parameter and returns a pointer to an object of `CBaseMtmUi`-derived class (i.e., a UI MTM).
- `CMtmUiDataRegistry::NewMtmUiDataLayerL(const TUid& aMtmTypeUid)` — Takes a message type UID reference as a parameter and returns a pointer to an object of `CBaseMtmUiData`-derived class (i.e., a UI data MTM).

The pointer to an abstract class, which is returned by all three methods, is sufficient for many tasks, but if access to specific methods available in the concrete classes is required, then more work needs to be done.

Do not try to delete the registry object immediately after obtaining the MTM. A registry must exist from the instantiation of the MTM to its deletion. Similarly, a valid Messaging Server session is required at all times when a registry exists. Also the objects must be deleted in the reversed order of creation: the MTM must be deleted first, then the registry, and finally the Message Server session.

4.4 Base Client MTM API

After the `CBaseMtm` class object has been obtained, it can be used for various tasks that do not require the built-in user interface components of Messaging to be invoked, such as creating, modifying, or sending a message without user interaction. Client MTMs use the concept of *current context*. It means that after establishing the client MTM object it is necessary to set the context of the MTM to point to the entry to be manipulated.

There are two methods for this purpose:

- `void SetCurrentEntryL(CMsvEntry* aEntry)` — Sets the context to an entry that is pointed to by a `CMsvEntry`-class object.
- `void SwitchCurrentEntryL(TMsvId aId)` — Sets the context to an entry identified by a `TMsvEntryId` type parameter.

Moreover, there are two other methods related to the context:

- `CMsvEntry& Entry()` — Returns a reference to the entry that is the current context. This function does not transfer the ownership.
- `TBool HasContext()` — Returns a Boolean value indicating if the context for the MTM has been established.

Other useful high-level functions of MTMs are:

- `virtual CMsvOperation* ReplyL(TMsvId aDestination, TMsvPartList aPartlist, TRequestStatus& aCompletionStatus)` — Creates a message that is a reply to the current (context) message. The exact function is message type-specific, but generally it will create a new message with the sender of the original message added as a recipient. For some MTMs (such as Bluetooth), a reply message cannot be created because the sender of the original message cannot be identified.
- `virtual CMsvOperation* ForwardL(TMsvId aDestination, TMsvPartList aPartlist, TRequestStatus& aCompletionStatus)` — Creates a message that has the same content as the current message.

Both the `ReplyL()` and `ForwardL()` functions *create* a new message. They do not send anything. Sending usually means that the message has to be copied from the local service to the message type-specific service (an exception to this scheme is the Bluetooth MTM, which does not have its own service entry). Both functions require three parameters and return an operation:

- The `aDestination` parameter is of `TMsvId` type and contains the entry ID of the destination — the folder in which the new message will be created.
- The `aPartlist` parameter is a `TMsvPartList` variable, used to specify which message parts (body, attachments, etc.) will be included in the new message.
- The third parameter, `TRequestStatus`, is notified when the operation completes (successfully or otherwise).

These functions are asynchronous: the new message is not ready immediately, but only when the operation returned from the function is complete.

Most client MTM functions require not only that the context be established but also that the entry be loaded. If a function accesses the content of an entry, the `LoadMessageL()` method must be called first; to commit any changes, a call to `SaveMessageL()` is required. Note that the methods `LoadMessageL` and `SaveMessageL` are not limited in their function only to messages. For instance, if an MTM function that modifies service settings is used, the application has to switch the context to the service entry, “load message,” make the modifications, and then “save message,” to update the changes.

Other useful methods from the client MTM deal with addressing a message:

- `const CDesCArray&` — Returns the array of message addressees.
- `void AddAddresseeL(const TDesC& aRealAddress)` — Adds an addressee to a message.
- `void RemoveAddressee(TInt aIndex)` — Removes an addressee at the specified index into the array returned by the `AddresseeList()` method.

These functions apply to the drafting of outgoing messages. An addressee is added using a simple text descriptor type; this allows the function to be generic enough for use with any message type. However, it makes it impossible to perform any address type validation because different message types require different address formats. Therefore, unlike the other client MTM methods that can be used regardless of the message type, this one requires knowledge of the message type in use and the type of address that should be used.

The following group of methods deals with the message subject and body:

- `inline const CRichText& Body()` and `inline CRichText& Body() const` — Return the body of a message as a rich text object. Note that there are two overloads of this method: one that returns a constant reference and another that returns a reference to a modifiable object.
- `virtual void SetSubjectL(const TDesC& aSubject)` — Takes a descriptor parameter and sets it as a subject of the current message.
- `virtual const TPtrC SubjectL() const` — Returns the subject as a constant descriptor pointer.

The latter two methods make sense only for message types that support a subject; should an application try to set the subject on a message that does not support a subject, it leaves with `KErrNotSupported`.

To avoid a situation where inappropriate action is taken with an MTM, the function `virtual TInt QueryCapability(TUId aCapability, TInt& aResponse)` allows an application to check what the MTM can and cannot do. The function takes two parameters: `aCapability` signifies the ID of the capability to be checked.

For example, to check that an MTM supports message subjects, `KUIdMtmQuerySupportSubject` (from `MtmUids.h`) is passed as the first parameter. Similarly, `KUIdMtmQuerySupportAttachments` is used to check if the MTM supports attachments and `KUIdMtmQuerySupportedBody` is used for support of the message body. The function returns `KErrNone` if the MTM has the capability; otherwise, `KErrNotSupported` is returned. The `aResponse` parameter is a reference to a `TInt`. If the capability is not supported, upon return that integer will contain a resource identifier. The resource is a string containing user interface text that can be used to show a more verbose explanation to a user.

The set of client MTM methods for attachment management follows:

- `virtual void CreateAttachmentL(TMsvId& aAttachmentId, TFileName& aDirectory)` — Creates a new attachment entry. The attached file is not copied. The application is responsible for placing the attached file in the folder returned in `aDirectory`. Upon return, the `aAttachmentId` parameter is used to notify the ID of the new attachment entry.
- `virtual void DeleteAttachmentL(TMsvId aMessageId, TMsvId aAttachmentId)` — Deletes an attachment specified by the message ID and attachment ID.

The final group of MTM methods deals with MTM-specific function invocation:

- `virtual void InvokeSyncFunctionL(TInt aFunctionId, const CMsvEntrySelection& aSelection, TDes8& aParameter)` — Calls the synchronous function identified by the first parameter (`aFunctionId` function ID) on one or more entries specified in the `aSelection` parameter with a function-specific parameter passed as a `TDes8` type reference.
- `CMsvOperation* InvokeAsyncFunctionL(TInt aFunctionId, const CMsvEntrySelection& aSelection, TDes8& aParameter, TRequestStatus& aCompletionStatus)` — Similar to the above, but invokes an asynchronous function and returns an operation. In addition to the three parameters identical to the function above, this one takes a `TRequestStatus` reference, which is notified when the request completes.

These invocation functions are specific to the message type, so the methods cannot be used in a generic way — the developer must know exactly what message type is going to be used and what functions it supports. Generally, the synchronous and asynchronous functions cannot be used interchangeably, because every function ID can only be used with one of these methods.

Finally, it is worthwhile to mention the `CreateMessageL` method in the base client MTM API. This method creates a new message entry accepting a single `aServiceId` parameter, which is the ID of the service under which the message will be created. Normally the messages are created under the local service.

The methods described up to this point are sufficient for most, but not all, tasks related to sending. For instance, the message settings such as CC recipients are message type-specific, and thus there is no common method for manipulating them in the base client MTM class. For example, `CreateAttachmentL()` will not be sufficient in many applications that may require more specific ways of tuning the attachment properties (for example, the character set identifier). In these situations, it will be necessary to use methods that are declared and implemented in the concrete classes derived from `CBaseMtm`.

4.5 Using the MTM UI

The client MTM API may be too low level for the application in question, the next option is to utilize the built-in system UI components for a message type. These components are provided by the UI MTM and supported by the UI data MTM base classes `CBaseMtmUi` and `CBaseMtmUiData`. The two classes are complementary and are usually used together because methods in the UI MTM may or may not be supported. By using the corresponding method in the UI data MTM, an application can inquire if the method is supported before attempting to invoke it.

Both classes have the `Type()` method, which returns the UID of the message type associated with the MTM. However, if the MTM has been instantiated using the registry objects, the UID will already be known.

As with the client MTM, most methods in the UI MTM depend on the context, or the current entry. However, there is no separate method for setting the current entry of the UI MTM. The UI MTM is always coupled with the client MTM, and the reference to the client MTM can be retrieved by calling

`inline CBaseMtm& CBaseMtmUi::BaseMtm()`. Once this is done it will be possible to use the context-setting methods of the client MTM (or any other method if need be).

The most common action is to create a new entry; however, unlike `CreateMessageL()` in the client MTM API, which only creates a new empty entry structure, the `virtual CMsgvOperation* CreateL(const TMsgvEntry& aEntry, CMsgvEntry& aParent, TRequestStatus& aStatus)` of the UI MTM opens the user interface for creating a new message. In practice, for most message types, it means opening the message editor. The exception is Bluetooth where a simple recipient selection dialog is opened.

The function takes three parameters:

- The `aEntry` parameter is of `TMsgvEntry` type and defines the entry to be created.
- The `aParent` parameter is a `CMsgvEntry` pointer that indicates the parent of the new entry to be created.
- The `aStatus` parameter is a `TRequestStatus` reference, which will be notified about the completion of the operation.

The function returns a `CMsgvOperation` pointer (note that most functions in the UI MTM API are asynchronous; details can be found in the header file). This method does not require the context to be set to any particular entry. Before invoking this method, it is necessary to confirm that the entry can be created with the intended parameters by calling `virtual TBool CanCreateEntryL(const TMsgvEntry& aParent, TMsgvEntry& aNewEntry, TInt& aReasonResourceId) const` in the UI data MTM. The parameters are the index entries of the parent and the new entry. This and other query methods in the UI data MTM return a Boolean value indicating whether or not the operation is supported; if it is not, the ID of the text resource with verbose explanation is returned via the `aReasonResourceId` parameter.

A major set of functions in the UI MTM API are related to opening, editing, viewing, and canceling entries. The methods are:

- `virtual CMsgvOperation* OpenL(TRequestStatus& aStatus)` — A generic open operation; in most cases either the `EditL()` or `ViewL()` operation will be invoked depending on the context.
- `virtual CMsgvOperation* EditL(TRequestStatus& aStatus)` — Starts editing the current entry. The semantics of this operation depend on the context. If the current entry is a message, the message editor will be launched for the message. If the entry is a service entry, this will have the effect of launching the message configuration settings dialog, as shown in Figure 4.
- `virtual CMsgvOperation* ViewL(TRequestStatus& aStatus)` — Again, the semantics of this operation depend greatly on the specific MTM and the context. The most common use of this method is to launch the built-in viewer for a message.
- `virtual CMsgvOperation* CancelL(TRequestStatus& aStatus, const CMsgvEntrySelection& aSelection)` — Cancels sending a message that has been submitted to the Outbox. Usually canceling a message does not involve any user interface interaction. It might seem that this method would be available through the client MTM API; however it belongs to this class.



Figure 4: Calling the `EditL()` method on the SMS service entry brings up the SMS configuration dialog

Note that each of these functions, except `Cancel()`, has two overloads: one that works on the current entry and another one that accepts a parameter of `CMsvEntrySelection` reference type. The latter can be used to invoke an operation on more than one entry. However, since the Nokia 7710 smartphone Messaging application permits launching only a single message editor or viewer at a time, specifying more than one message to these functions has no effect. Again, it is good practice before using any of the above-mentioned methods to query the UI data MTM to check that the intended usage is supported. The methods for making these checks are:

- `virtual TBool CanOpenEntryL(const TMsVEntry& aContext, TInt& aReasonResourceId) const,`
- `virtual TBool CanEditEntryL(const TMsVEntry& aContext, TInt& aReasonResourceId) const,`
- `virtual TBool CanViewEntryL(const TMsVEntry& aContext, TInt& aReasonResourceId) const,`
- `virtual TBool CanCancelL(const TMsVEntry& aContext, TInt& aReasonResourceId) const.`

Each of the methods takes a `TMsVEntry` object reference as the `aContext` parameter. They use the same protocol for checking the capability as the `CanCreateEntry()`.

The following set of functions is used for replying and forwarding messages:

- `virtual CMsvOperation* ReplyL(TMsVId aDestination, TMsVPartList aPartList, TRequestStatus& aCompletionStatus)` — Similar to the corresponding method in the client MTM API, this method creates a reply message, but differs because it launches the message editor to allow the user to create the message.
- `virtual CMsvOperation* ForwardL(TMsVId aDestination, TMsVPartList aPartList, TRequestStatus& aCompletionStatus)` — Analogous to the `ForwardL()` method of the client MTM, but again this one also opens the editor.

As with all the edit and view methods, a check should be made with the UI data MTM to determine if the operation is supported. This is done using the following methods:

- `virtual TBool CanReplyToEntryL(const TMsVEntry& aContext, TInt& aReasonResourceId) const` — Takes an `aContext` parameter, which indicates the entry to be replied to.
- `virtual TBool CanForwardEntryL(const TMsVEntry& aContext, TInt& aReasonResourceId) const` — For forwarding.

Again, the same protocol is used to inform whether the operation is supported for this MTM and context.

Another set of methods is available for sending and receiving messages. These functions are:

- `virtual CMsvOperation* CopyToL(const CMsvEntrySelection& aSelection, TRequestStatus& aStatus),`
- `virtual CMsvOperation* MoveToL(const CMsvEntrySelection& aSelection, TRequestStatus& aStatus),`
- `virtual CMsvOperation* CopyFromL(const CMsvEntrySelection& aSelection, TMsVId aTargetId, TRequestStatus& aStatus),`
- `virtual CMsvOperation* MoveFromL(const CMsvEntrySelection& aSelection, TMsVId aTargetId, TRequestStatus& aStatus).`

The functions of the `CMsvEntry` class can be used for copying or moving entries within the local service. The special case is copying or moving to or from a remote service entry, which has the effect of sending or receiving messages. (Bluetooth MTM is an exception because it does not have a service. Sending is performed by calling `InvokeAsyncFunctionL()`.) Each of the functions expects that the context be set to the service entry to or from which the message will be sent or received. Each function takes the `aSelection` parameter, an array of message entry IDs for the messages that are to be sent or received. Moreover, `CopyFromL()` and `MoveFromL()` take the ID of the folder in which to receive, as the `aTargetId` parameter. The `aStatus` parameter is the usual `TRequestStatus` reference, and the function returns a `CMsvOperation` pointer like all other asynchronous functions when it completes.

Although MTMs usually support sending, very few actually support receiving through this interface. For most message types, automatic receiving of messages is performed by the underlying operating system. An application cannot force, for example, fetching a text message through these APIs.

The UI data MTM has the following methods to check if the planned operation is supported:

- `virtual TBool CanCopyMoveToEntryL(const TMsVEntry& aContext, TInt& aReasonResourceId) const` — Checks if `CopyToL()` or `MoveToL()` operations can be performed for an entry specified by the `TMsVEntry` reference parameter.
- `virtual TBool CanCopyMoveFromEntryL(const TMsVEntry& aContext, TInt& aReasonResourceId) const` — Checks if `CopyFromL()` or `MoveFromL()` operations can be performed for an entry specified by the `TMsVEntry` reference parameter.

As before, the usual protocol is used to return a response.

The following set of methods is related to deleting entries. The functions in this group are:

- `virtual CMsvOperation* DeleteFromL(const CMsvEntrySelection& aSelection, TRequestStatus& aStatus)` — Asynchronous function that accepts an entry selection parameter. It deletes the entries that are under the current context, which must be a folder or service of the current MTM.
- `virtual CMsvOperation* UnDeleteFromL(const CMsvEntrySelection& aSelection, TRequestStatus& aStatus)` — Undoes the deletion performed by the previous method.
- `virtual CMsvOperation* DeleteServiceL(const TMsVEntry& aService, TRequestStatus& aStatus)` — Deletes a service entry. Obviously not every service entry can be deleted. For instance, an e-mail account may be deleted, but an SMS service cannot be, because one must exist.

As with the other methods described in this section, these functions are asynchronous and return a pointer to an operation object.

The corresponding functions in the UI data MTM are:

- `virtual TBool CanDeleteFromEntryL(const TMsVEntry& aContext, TInt& aReasonResourceId) const` — Used before calling `DeleteFromL()`.
- `virtual TBool CanUnDeleteFromEntryL(const TMsVEntry& aContext, TInt& aReasonResourceId) const` — Used before calling `UnDeleteFromL()`.
- `virtual TBool CanDeleteServiceL(const TMsVEntry& aService, TInt& aReasonResourceId) const` — Used before calling `DeleteServiceL()`.

The final group of methods is very similar to those present in the client API:

- `virtual TInt QueryCapability(TUId aCapability, TInt& aResponse)` — Interrogation method that allows a check to be made on the capabilities of the MTM before attempting to use them. The difference is that this method is used to check the user interface capabilities.
- `virtual void InvokeSyncFunctionL(TInt aFunctionId, const CMsVEntrySelection& aSelection, TDes8& aParameter)` — Almost identical to the method in the client MTM API, it can be used to invoke MTM-specific synchronous user interface operations. It is also possible to invoke the client MTM methods in this way, and the UI MTM will delegate the call to the client.
- `virtual CMsvOperation* InvokeAsyncFunctionL(TInt aFunctionId, const CMsVEntrySelection& aSelection, TRequestStatus& aCompletionStatus, TDes8& aParameter)` — Similar to the method above except that it invokes an asynchronous operation.

If an application uses UI and UI data MTMs frequently, instead of using the registries every time to load and unload the MTM objects, it is more efficient to use the `CMtmStore` class. This class has built-in mechanisms for caching and dynamic loading and unloading of UI and UI data MTMs. To instantiate one of the MTMs all that is required is a session with the Messaging Server. The reference to the `CMsvSession` object is passed to `CMtmStore::NewL()` instantiation method. Once the store is constructed, the following methods are available:

- `CBaseMtmUiData& MtmUiDataL(TUId aMtm)` — Returns a reference to a UI data MTM object. The ownership remains with the store. The method is passed the `aMtm` UID parameter indicating which message type is needed.
- `CBaseMtmUi& ClaimMtmUiL(TUId aMtm)` — Gets an instance of the UI MTM corresponding to the UID, which is passed as the sole parameter. The ownership remains with the store. However, once the MTM is claimed, no other caller can obtain it as it is locked. It remains locked until it is released.
- `CBaseMtmUi& ClaimMtmUiAndSetContextL(const TMsVEntry& aContext)` — Similar to `ClaimMtmUiL()` but also sets the context to the entry that is passed as the `aContext` parameter. The MTM UID is inferred from the entry.
- `void ReleaseMtmUi(TUId aMtm)` — Releases the MTM with the given UID after it has been claimed.
- `CBaseMtmUi& GetMtmUiLC(TUId aMtm)` — Does the same as `ClaimMtmUiL()` but also places a releasing item on the cleanup stack. If a leave should occur, the cleanup entry ensures that the MTM is released. `CleanupStack::PopAndDestroy()` can be used to release the MTM that obtained in this way.
- `CBaseMtmUi& GetMtmUiAndSetContextLC(const TMsVEntry& aContext)` — Does the same as `GetMtmUiLC()` but also sets the context to the entry specified by the `aContext` parameter.

All methods and classes described so far can be applied to any message entry, but are mostly targeted at outgoing messages. By using these methods, messages can be composed and sent, and received messages opened or otherwise examined. However, in general, messages are not fetched using a client application. Even though some MTMs (for example, e-mail) may support application-initiated fetching of messages, in most cases the messages arrive automatically in the device.

However, an application may need to know when messages arrive, or when anything new appears or changes in the messaging structures. This can be achieved by using *observers*. Two classes can be used. The first is `MMSvSessionObserver`, which is described in the Messaging Server session initialization in Section 4.1, “Messaging Server Session.” This interface is notified when anything within the Messaging Server changes, for example, if entries are created, modified, or deleted, or MTMs are installed or uninstalled.

It is possible to control whether or not an entry event is fired by calling the `SetReceiveEntryEvents()` method, which takes a single Boolean parameter to turn the receive on (`ETrue`) or off (`EFalse`). For typical purposes, being notified about every server event can be overwhelming, and an application may only need to be notified about events that occur within a selected entry. The interface that such an observer has to implement is `MMSvEntryObserver` with its single method `void HandleEntryEventL (TMSvEntryEvent aEvent, TAny* aArg1, TAny* aArg2, TAny* aArg3)`. The `aEvent` parameter for this function is an event type enumeration that can indicate an entry changed; children added, deleted, or changed; or the whole entry deleted or moved.

The other three parameters are generic `TAny` pointers and their meaning depends on the specific event. A detailed list of session events and their related parameters is available in the Symbian OS v7.0s documentation accompanying the Nokia 7710 device SDK (see the `MMSvSessionObserver` class). To register an object for entry notifications, call `CMsvEntry::AddObserverL()`; deregister by calling `CMsvSession::RemoveObserver()`.

5 Conclusion

This document began by discussing the Nokia 7710 smartphone's messaging subsystem and the abstract Send As APIs, which are sufficient for most simple message functionality in an application. However, the messaging subsystem offers developers much more control with the low-level MTM API, and information on this topic was covered in subsequent sections of the document.

Having read the document, developers should now be able to design applications that create, send, and receive any of the message types supported by the Nokia 7710 smartphone, both without the user's intervention and by providing the user with control over the message type and content.

The second document in this series takes a closer look at the messaging APIs for manipulating multimedia messages in *Nokia 7710: Using The Messaging APIs For MMS*.

6 Terms and Abbreviations

Term or abbreviation	Meaning
MTM	Message Type Module. A software plug-in that is responsible for handling a single message type, for example SMS or MMS.

7 References

Nokia 7710: Using The Messaging APIs For MMS

Nokia 7710: Creating MMS Content

8 Evaluate This Document

In order to improve the quality of documentation, we kindly ask you to fill in the [document survey](#).