

USING THE NOKIA 3650 MESSAGING API

Version 1.0

06 Sept 02

Table of Contents

1. INTRODUCTION	3
1.1 PURPOSE AND SCOPE	3
1.2 HARDWARE AND SOFTWARE REQUIREMENTS	3
1.3 DEMONSTRATION SOURCE CODE AVAILABILITY	4
2. MMS OVERVIEW	4
3. COMPOSING A MULTIMEDIA MESSAGE	5
3.1 OVERALL MESSAGE STRUCTURE	5
3.2 FINDING THE SERVICE ID.....	5
3.3 INITIALIZING THE MMS CLIENT MTM.....	6
3.4 CREATING A MESSAGE	7
4. MANIPULATING MMS MESSAGES	8
4.1 LOADING AND READING MESSAGES.....	8
4.2 SAVING MESSAGES.....	9
4.3 MOVING MESSAGES.....	9
4.4 DELETING MESSAGES	9
4.5 CHANGING THE SUBJECT AND RECIPIENTS	10
4.6 UNDER-CONSTRUCTION FLAGS	10
4.7 WORKING WITH ATTACHMENTS	10
5. SENDING A MULTIMEDIA MESSAGE	11
6. CONCLUSION	11

Change history

06 Sept 02	Version 1.0	Document published in Forum Nokia.
------------	-------------	------------------------------------

Disclaimer:

Nokia Corporation disclaims all liability, including liability for infringement of any proprietary rights, relating to the 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.

1. INTRODUCTION

The Nokia 3650 imaging phone is based on the open Series 60 Platform standard and includes a built-in camera capable of capturing images at resolutions up to 640 x 480 pixels with a pallet of up to 16 million colors. Camera services are implemented as a shared resource that can be accessed simultaneously by several applications.

1.1 Purpose and Scope

The following tutorial is intended for application developers who want to make use of the Multimedia Messaging Service (MMS) capabilities built into Nokia 3650, a Series 60 Platform device. This document presents a brief introduction to accessing the messaging API on the handset to:

- 1) Connect to the messaging server and compose a multimedia message
- 2) Store, move, and retrieve a multimedia message
- 3) Send or queue for sending a multimedia message

1.2 Hardware and Software Requirements

The code snippets included in this tutorial are extracted from a fully functional source code example contained in file `3650MMEExercise.cpp` available for download at www.forum.nokia.com. To successfully run the example code found in this tutorial, developers should have a system that meets various hardware and software requirements. These include:

- 1) Operating system – Microsoft Windows NT 4.0, Service Pack 6 or Windows 2000. Other operating systems may or may not work with these examples, but these and more recent Microsoft platforms are likely to yield the best results.
- 2) A minimum of 500 megabytes free hard disk space.
- 3) The Series 60 SDK for the Symbian operating system. This can be downloaded from Nokia.com or obtained via a Nokia-supplied CD.
- 4) Microsoft Visual C++ 6.0 with Service Pack 3 – this is strongly recommended, especially if developers expect to build and debug code.
- 5) Finally, the system should meet the processor speed and RAM requirements called for by Microsoft Visual C++.

Final testing requires access to a Nokia 3650 handset. Executable packages are transferred to the phone via an infrared link or other means. Therefore, the development system must have one of these transfer methods available. For a complete description of the SDK, the simulator, and the transfer methods, please see the documentation included in the SDK download available at www.forum.nokia.com/tools.

1.3 Demonstration Source Code Availability

A source file, `3650mmsexercise.cpp`, and other files that accompany this tutorial are available for download. This code provides the basis for building an MMS application using Nokia 3650. Among other things, the application provides a menu option and accompanying function to create and save an MMS message. There is also a menu/function pair to move a message from one folder to another and to send the first message in the Outbox. The code has several functions that illustrate MMS capabilities and operation. These are commented out in the source file but can be uncommented for instructional purposes. These include:

- `InstantiateMmsMtmL()`
- `HandleCommandL(TInt aCommand)`
- `CreateNewMessageL()`
- `CreateNewMessageL(TMsvId aServiceId, TMsvId aFolder)`
- `MoveMessageL(TMsvId aFrom, TMsvId aDestination)`
- `LoadMessageL(TMsvId aFolder)`
- `ModifyMessageL(TMsvId aFolder)`
- `SendMessageL()`
- `OpenFileL(const TDesC& aFileName)`

2. MMS OVERVIEW

As the name suggests, MMS involves the sending and receiving of multimedia messages. The messages can originate from and terminate on another handset or an application running on another type of device. Because of their multimedia nature, the messages can consist of graphics, text, audio, and animation. This provides end users with a more interesting and enjoyable experience.

Nokia 3650 is built on Series 60 Platform, and the handset offers a number of MMS capabilities. These include message composition, transmission, and manipulation. The implementation of MMS in the handset is done through a messaging server/client model. The server runs in kernel mode and provides support to client applications. These services include managing access to message data and the delegation of requests to server side Message Type Modules (MTMs).

Nokia 3650 can receive multiple-slide MMS messages and can send single-slide messages. There can only be one type of multimedia element – text, image, audio, or video – per slide. This constraint can be circumvented through the use of the Synchronized Multimedia Integration Language, or SMIL, to orchestrate the use of multiple slides. However, developers should keep in mind that there is no timing possible using this approach.

The viewer on Nokia 3650 supports MMS basics such as JPEG, GIF, and WBMP formats. It also supports progressive JPEGs, PNG, WAV, MIDI, MPEG4, RealVideo, and other formats. The handset

does not support streaming video for various technical reasons, such as the lack of infrastructure bandwidth. The transfer of large attachments and messages is done in the background by the handset without needing end-user attention or intervention.

3. COMPOSING A MULTIMEDIA MESSAGE

MMS messaging is asynchronous in nature and consumes shared resources on the handset. Because of this, Nokia 3650 and Series 60 Platform use a client/server messaging architecture. MMS applications for the handset, therefore, must initialize a client and connect to the messaging server. All message data manipulation should be done through this server. Using other avenues, such as direct file operations, may lead to unexpected and unwanted results. This section describes how to connect to the message server and compose a message.

3.1 Overall Message Structure

On Nokia 3650, messages are stored in a standard file structure. The root is at `KMsvRootIndexEntryId`, with the services branching off from that root. The local service, for example, is at `KMsvLocalServiceIndexEntryId`. Below that are folders for Draft (`KMsvDraftEntryId`), Inbox (`KMsvGlobalInBoxIndexEntryId`), Outbox (`KMsvGlobalOutBoxIndexEntryId`), and Sent (`KMsvSentEntryId`) messages. Messages themselves are stored as entries in those folders, so that a particular draft message might be identified as `0x1234abcd`. An index file managed by the message server links message identifiers to the correct message file.

3.2 Finding the Service ID

Nokia 3650 supports five different message types through the use of SMS, MMS, POP3, IMAP4, and SMTP client side MTMs. These are derived from the `CBaseMtm` class of the Symbian OS messaging system. Each MTM client provides the additional interface and capabilities needed for operations relating to that specific message type. These include address list manipulation, creation of forward and reply messages, message subject access, message validation, and access to the message store.

An important piece of information is the numeric identifier for a particular service. This is used as an input parameter in various functions and impacts the storage place and access path for any message. The service ID may not be the same from one handset to another. One handset, for example, may have had more or fewer services installed. The following code returns the service ID by searching through the root of the message structure. In this particular example it is set to find a POP3 service, but the code can be modified to look for any desired messaging channel. This example also implements the out-of-resource and exception handling found in good Symbian programming:

```
// select the root index to start the search
```

```
CMsvEntry* currentEntry =
iMsvSession.GetEntryL(KMsvRootIndexEntryId);

CleanupStack::PushL(currentEntry);

// don't sort the entries

currentEntry->SetSortTypeL(TMsvSelectionOrdering(KMsvNoGrouping,
EMsvSortByNone, ETrue));

TInt count=currentEntry->Count();

// loop for every child entry of the root index
for(TInt i = 0;i<count;i++)
    {
        const TMsvEntry& child = (*currentEntry)[i];

        // is the current child the same type as the type we are
        looking for ?

        if (child.iMtm == KUidMsgTypePOP3)
            {
                { // yes it is. Get the service id and break out of loop.
                    break;
                }
            }
    }

CleanupStack::PopAndDestroy(currentEntry);
```

3.3 Initializing the MMS Client MTM

Before using the MMS client MTM, an application must get a pointer to the `CMmsClientMtm` object instance. The returned value is then used when creating messages, interacting with the message server, and for various other operations.

Getting the pointer first requires the implementation of an interface to the MMS session observer. The following code accomplishes this:

```
CMsvSession* iMsvSession;
CClientMtmRegistry* iMtmReg;
CMmsClientMtm* iMmsMtm;
```

The next step opens a connection to the message server. Note the use of a *leave* for exception handling, a highly recommended Symbian programming practice denoted by the `L` at the end of functions such as `OpenAsyncL()`. The returned value from `OpenAsyncL()` is a pointer to a

session connected to the message server. The `CClientMtmRegistry` acts as a factory, which creates an instance of the `CMmsClientMtm` object (`iMmsMtm`) specified by the `KUIdMsgTypeMultimedia`.

```
iMsvSession = CMsvSession::OpenAsyncL(*this);  
  
iMtmReg = CClientMtmRegistry::NewL(iMsvSession);  
  
// get a pointer to the multimedia client mtm  
  
iMmsMtm =  
static_cast<CMmsClientMtm*>(iMtmReg->NewMtmL(KUIdMsgTypeMultimedia));
```

The final step completes the initialization by implementing a callback function. This must be done because the message server session may need to alert the client when certain events occur, such as when a session is open and ready to be used. The code below accomplishes this:

```
CMyAppUi::HandleSessionEventL(TMsvSessionEvent aEvent, ...) { ... }
```

3.4 Creating a Message

With the client initialized and the service ID known, an application can then create a new message. This can be done in one of two acceptable ways. The first makes use of the service ID and the `CreateMessageL()` function. The second specifies where in the message store the new message is to go by using the `CreateNewEntryL()` function.

The first approach is given in the code below. It is simpler because many of the housekeeping chores associated with message creation are handled automatically:

```
CMsvEntry* entry = iSession-  
>GetEntryL(KMsvGlobalOutBoxIndexEntryId);  
  
CleanupStack::PushL(entry);  
  
iMmsMtm->SwitchCurrentEntryL(entry->EntryId());  
  
CleanupStack::PopAndDestroy(); // entry  
  
TMsvId serviceId = iMmsMtm->DefaultSettings()  
  
iMmsMtm->CreateMessageL(serviceId);
```

In the second approach, the service ID is not needed:

```
TMsvId dest = KMsvGlobalOutBoxIndexEntryId;

CMsvOperationWait* wait = CMsvOperationWait::NewLC();

wait->Start();

CMsvOperation* operation = iMmsMtm->CreateNewEntry(dest, wait-
>iStatus);

CleanupStack::PushL(operation);

CActiveScheduler::Start()

if (wait->iStatus.Int() != KErrNone) { //handle error }

TPckgBuf<TMsvId> pkg;

pkg.Copy( operation->ProgressL());

TMsvId indexEntry = pkg();

CleanupStack::PopAndDestroy(2); // operation and wait

iMmsMtm->SwitchCurrentEntryL(indexEntry);

iMmsMtm->LoadMessage();
```

4. MANIPULATING MMS MESSAGES

Once an MMS message has been created, an application may need to edit it, send it to another recipient, add an attachment, or otherwise manipulate it. The `CMmsClientMtm` class provides the means to accomplish all of these actions. This section outlines the methods for manipulating an MMS message.

4.1 Loading and Reading Messages

Prior to altering the makeup or properties of an MMS message, an application will typically first load and display the message. This is done by setting the context within the client MTM to the desired message and then calling the `LoadMessageL()` function, as illustrated by the code below:

```
iMmsMtm->SwitchCurrentEntryL(desiredEntryId);

iMmsMtm->LoadMessageL();

const TPtrC sender = iMmsMtm->Sender();

const CDesCArray& addresses = iMmsMtm->AddresseeList();
```

NOKIA

Using the Nokia 3650 Messaging API

Version 1.0

4.2 Saving Messages

In order to save messages, call the `SaveMessageL()` function. This should be done after creating a new message or modifying an existing message and prior to working on any other message. Failure to follow this protocol will result in the loss of all work and modifications. The code to save a message is given below:

```
iMmsMtm->SaveMessageL();  
  
iMmsMtm->SwitchCurrentEntryL(newEntryId);
```

4.3 Moving Messages

The following code moves a message to another folder. In this example, the operation moves the message to the Draft folder. The destination can be changed by changing the parameters of the `MoveL()` function:

```
cEntry->SetSortTypeL(TMsvSelectionOrdering(KMsvNoGrouping,  
EMsvSortByNone, ETrue));  
  
CMsvOperation* myOperation = NULL;  
  
TMsvId currentId = iMmsClient->Entry().EntryId();  
  
CMsvOperationWait* wait = CMsvOperationWait::NewLC();  
  
wait->Start();  
  
// move to Draft  
  
TRAPD(aError, myOperation = cEntry->MoveL(currentId,  
KMsvDraftEntryId, wait->iStatus));  
  
if ( aError != KErrNone )  
    { .... }
```

4.4 Deleting Messages

The index file of the message store keeps track of message IDs and automatically links these to the appropriate message file. For that reason, do not use a file operation to delete a message directly. Doing so will corrupt the index and create problems.

Instead, use `CMsvEntry::DeleteL()` to delete a message. This method deletes all the resources under the message entry including any attachment files. If a message is locked because it is in use by another application or by the system, use `CMsvSession::RemoveEntry(entryId)`. Doing so marks the message for deletion later when it is unlocked.

4.5 Changing the Subject and Recipients

The subject of a message is stored in `iDescription` of a `TMsvEntry` object. Although `iDescription` is a public data member of the class, do not modify it directly. Instead use the code below to set the subject:

```
CMmsClientMtm::SetSubjectL();  
  
CMmsClientMtm::SetMessageDescriptionL();
```

If the subject is not specified, the MMS MTM will use the first few characters from the first plain text attachment as the subject. If there is no plain text attachment, the subject will be blank.

As for recipients in the "To," "Cc," or "Bcc" fields, currently the MMS MTM only recognizes phone numbers and e-mail addresses. Each address type is recognized automatically. Setting the "To" field is done by `CMmsClientMtm::AddAddresseeL()`, while setting the other fields is accomplished by `CMmsClientMtm::AddTypedAddresseeL()`.

4.6 Under-Construction Flags

While changes are being made to a message, an application must mark it invisible and under construction. If this is not done, other applications will be able to see the message and may try to modify it simultaneously. In essence, the message should be locked. The following code does this:

```
TMsvEntry::SetInPreparation(ETTrue);    // message is under  
                                         // construction  
  
TMsvEntry::SetVisible(EFalse);         // message is invisible  
  
CMsvEntry::ChangeL()                   // commits the changes
```

The default settings for a new message created by calling `CMmsClientMtm::CreateMessageL()` are invisible and under construction. When complete, a new message should be set to visible and the under-construction flag removed before the message is saved to a local message store. This must be done because the system deletes all messages set to under construction at every boot up.

4.7 Working with Attachments

There are several methods to create an attachment. The typical method for creating an attachment from a file is:

```
CreateAttachment2L(TMsvId& aAttachmentId, const TDesC& aFullPath)
```

which is synchronous and handles all file manipulation automatically. The function creates an attachment message entry under the message server and tries to resolve and set the MIME content type for the attachment. The attachment file is copied by the message server, so an application can delete the original without corrupting the new multimedia message.

However, this approach can take a long time if the attachment is large. During that span, the synchronous procedure will keep the application from doing anything else. Therefore, another method, `CreateAttachmentL(TMsvId& aAttachmentId, TFileName& aDirectory)`, can be used. This approach provides a way to implement an asynchronous operation. To create an attachment using this tactic, call the function, and use an active object and an asynchronous file copy method. The disadvantage to this approach is that all file manipulation must be handled by the application itself.

A text attachment can be created by using several versions of `CreateTextAttachmentL(TMsvId& aAttachmentId, ...)`.

There are four functions that deal with the various aspects of extracting an attachment and related information from a message. `GetAttachmentsL()` returns the array of attachments connected to a message. `GetAttachmentPathL(TMsvId aAttachmentId, TFileName& aFilePath)` produces the full path to an attachment. `AttachmentNameL(TMsvId aAttachmentId)` yields the file name of an attachment without the path. Finally, `AttachmentTypeL(TMsvId aAttachmentId)` returns the MIME type of an attachment.

5. SENDING A MULTIMEDIA MESSAGE

Since an MMS message can be large and will be sent in the background, an application should check for errors before committing it to transmission. That can be done using `ValidateMessage(TMsvPartList aPartList)`, which tests the current message to see if it is ready to be sent and so avoids errors in the transmission phase of the process. The various message parts are defined in *mtmdef.h*, which is found in the Series 60 SDK for the Symbian operating system.

With the message complete and verified, an application then calls `SendL()`. This sends a message asynchronously to the MMS server via the MTM server. The message is first moved to the Outbox folder. After a successful transmission, the message is moved to the Sent folder if the current service settings call for that. Otherwise, the message is deleted. It is important to note that messages flagged read-only cannot be sent.

6. CONCLUSION

This tutorial provides basic information about the multimedia messaging capabilities of Nokia 3650. The complete example code for this tutorial is contained in the file `3650MMSExcercise.cpp` available at www.forum.nokia.com.

For more information about Series 60 Platform, please download the Series 60 Platform Information kit available at www.forum.nokia.com.