07: Client-Server and Files

Tutorial / Exercise

Goal

You will create a fully working audio player application for mp3/aac/...-files based on the S60 platform. To make this work, client-server communication is demonstrated through the use of the multimedia framework as well as the file server.

Introduction

When developing real-world applications for Symbian OS, you will almost certainly get in touch with an API that relies on the client-server framework. Examples include sockets, messaging or – as in this exercise – the multimedia framework and file access.

The first is used to show a higher level interface to an asynchronous service provider. With just a few lines of code, you can create your own audio player that automatically supports all of the codecs that are installed in the phone.

The file API is used to store and retrieve the last played track and demonstrates a more low-level interaction with a server.

Structure of this Exercise

The project for this exercise is the first UI application. Creating those is not directly covered by the Symbian Academy course materials. Consequently, this exercise consists of three parts:

1. ***User Interface (optional):*** Start with a new project and create the user interface yourself.
2. ***Multimedia Framework:*** Implement the communication to the audio player utility class.
3. ***File Access:*** Save and retrieve the last played track.

The first, optional part is presented as a tutorial and contains a description of the steps you have to complete in order to create a simple UI application based on the S60 UI-Designer of *Carbide.c++ Developer Edition*.

If you either do not want to create the UI yourself or if you are working with *Carbide.c++ Express* Edition or another IDE, start with part 2 and the supplied pre-written framework.

The two main parts of this exercise (2 & 3) can be done in two different ways:

- Through the tutorial in this document, if you did part 1 of this module yourself.
- Through the pre-written framework and the marked edit-positions, like in previous exercises.

Tutorial

Introduction

This project is based on S60 3rd Edition. The Symbian Academy course covers development for Symbian OS in general and doesn't describe how to work with specific UI implementations like S60 from Nokia and UIQ.

Therefore, a short tutorial explains how to create the UI application using the S60 UI designer from *Carbide.c++ Developer Edition*. However, the tutorial is only intended as an overview of how to create this specific user interface for the project and does not contain detailed explanations of the UI framework in general.

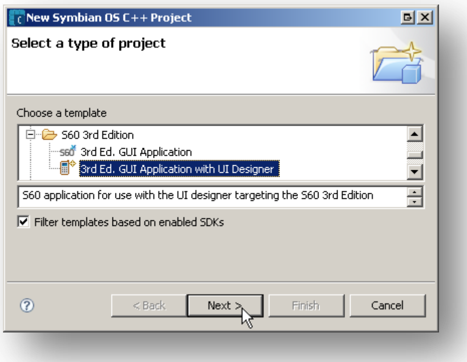
For more information on how to create applications for the S60 user interface, find tutorials at:

- **Forum Nokia:** http://www.forum.nokia.com/
- **symbianresources.com:** http://www.symbianresources.com

If you either do not have the *Developer Edition* (or better) of *Carbide.c++* or if you are not interested in how to create a user interface, you can skip the first part and start directly with the provided framework that contains the pre-written UI design. In this case, please start with the section called *"Part 2: The Client Side of the Multimedia Framework"* and/or work through the edits in the source code.

Part 1: Creating the Framework

Create a new application in *Carbide.c++ Developer Edition* (or better) and use the *“3rd Ed. GUI Application with UI Designer”*-template.



New Symbian OS C++ Project

Select a type of project

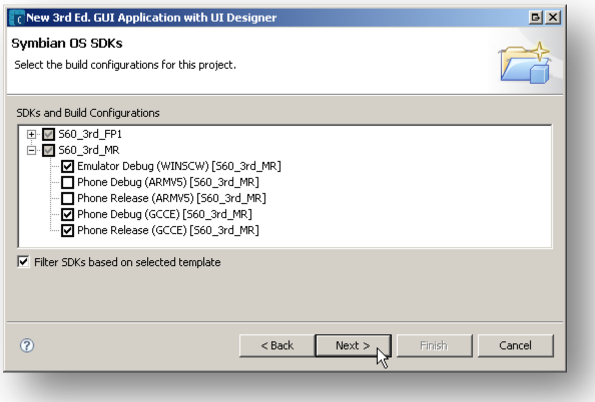
Choose a template

- S60 3rd Edition
  - 3rd Ed. GUI Application
  - 3rd Ed. GUI Application with UI Designer

S60 application for use with the UI designer targeting the S60 3rd Edition

☑ Filter templates based on enabled SDKs

Choose *“AudioPlayer”* as project name. On the next page, you should at least enable the *“Emulator Debug”* build configuration for the *“S60_3rd_MR”*-SDK. If you also want to deploy the project for the phone, you will most likely need the debug and release-configurations for the *GCCE*-compiler. It is free and is installed with the Symbian OS SDKs. The *ARMV5*-compiler is a commercial product; while it creates more optimized and smaller code than the free *GCCE* compiler, it is usually not necessary for 3rd party developers.



New 3rd Ed. GUI Application with UI Designer

Symbian OS SDKs

Select the build configurations for this project.

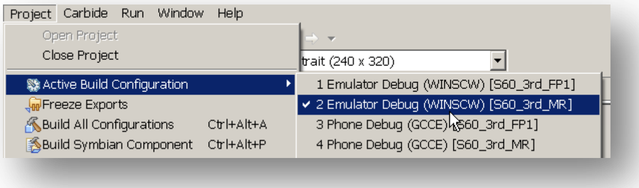
SDKs and Build Configurations

- ☑ S60_3rd_FP1
- ☑ S60_3rd_MR
  - ☑ Emulator Debug (WINSCW) [S60_3rd_MR]
  - ☐ Phone Debug (ARMV5) [S60_3rd_MR]
  - ☐ Phone Release (ARMV5) [S60_3rd_MR]
  - ☑ Phone Debug (GCCE) [S60_3rd_MR]
  - ☑ Phone Release (GCCE) [S60_3rd_MR]

☑ Filter SDKs based on selected template

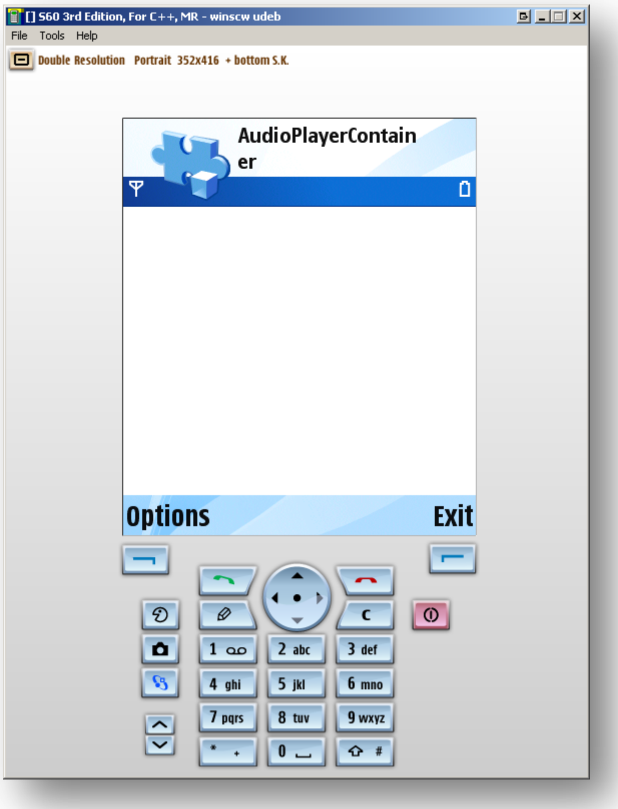
Continue to the *“S60 UI Design Selection”* screen. We will use the default, *“Empty”* template for the audio player. On the following page, *“Container Name and Type”*, make sure that *“Support view switching”* is enabled. While our application will only contain one view and wouldn’t need view switching, the architecture that’s generated by the wizard with this setting is better suited to separate UI and control logic.

Once your project is created, test it in the emulator. Make sure the proper build configuration for the emulator is active and launch the project.


Project
Carbide
Run
Window
Help
Open Project
Close Project
Active Build Configuration
Freeze Exports
Build All Configurations Ctrl+Alt+A
Build Symbian Component Ctrl+Alt+P
trait (240 x 320)
1 Emulator Debug (WINSCW) [S60_3rd_FP1]
✓ 2 Emulator Debug (WINSCW) [S60_3rd_MR]
3 Phone Debug (GCCE) [S60_3rd_FP1]
4 Phone Debug (GCCE) [S60_3rd_MR]

You should find the new application in the *"Installat."*-folder of the menu in the emulator. Keep in mind that S60 is not (yet) a touch screen UI, therefore you should navigate using the device keys at the bottom of the emulator.

This is what your application should initially look like.


[] S60 3rd Edition, For C++, MR - winscw udeb
File
Tools
Help
Double Resolution Portrait 352x416 + bottom S.K.
AudioPlayerContainer
Options
Exit
C
1
2 abc
3 def
4 ghi
5 jkl
6 mno
7 pqrs
8 tuv
9 wxyz
* +
0
#

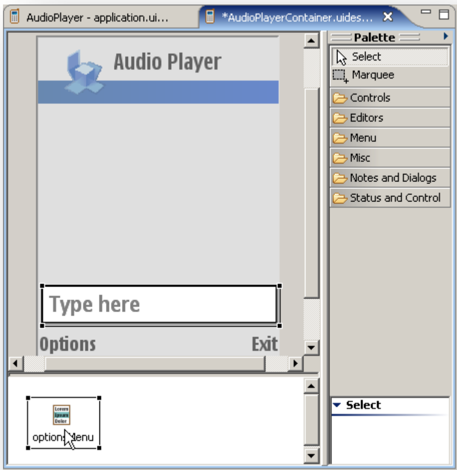
No menu is defined as of now; the *"Exit"*-button does already work. It's important that you always exit your application first and then close the emulator window, instead of simply directly closing the emulator. Like our code for the console applications did, the UI-framework of the emulator automatically compares the memory state before launching your application and after it was closed. If memory leaks are detected, you will be warned. It's always easier to identify and fix a memory leak right after you wrote the code that was causing it, than having to fix several leaks at the end, when a huge application is finished.

Adapting the Container

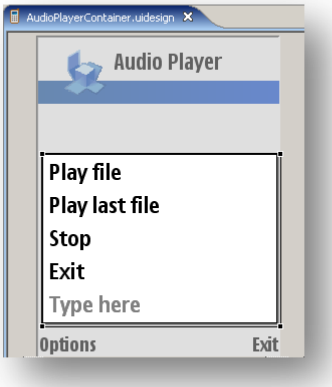
Now that we know that your application works, the next task is to adapt the look and functionality of the application to our needs. The `AudioPlayerContainer.uidesign` file should be open by default; if it isn't, open it by double-clicking on the file.

To make your application more personal, you should rename the title from `AudioPlayerContainer` to something better, like `Audio Player`. Do so by double-clicking on the title in the UI designer and then replace the text with the new content.

To create a new menu, left-click once on the `optionsMenu` icon below the screen:

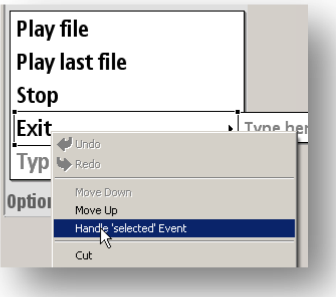


A menu will appear, where you can add your menu items by simply double-clicking on the *"Type here"*-text and writing the name of the new menu item. In total, create four new menu items (see the following screen shot) and save the UI design. This will make the designer automatically create the corresponding code.



Implementing the "Exit" menu command

In this case, the application can be closed through the right softkey; nevertheless, it's less confusing for the user if he can also exit through the options menu. As it's the easiest task, we will start with implementing this functionality. Right-click on the *"Exit"* menu command in your UI design and choose *"Handle 'selected' event..."*.



Whenever the user selects a menu command, the `HandleCommandL()` method of the current view will be called. If the application does not handle the event in this place, it'll be sent to the `AppUi`-class, which can be seen as the level above all views of the application.

The UI designer automatically created the code to handle the menu command in `HandleCommandL()` and defined a new function, where we can insert our code for the exit command. In this case, we will transform our custom command (which just has the name *"Exit"*) to a Symbian OS wide exit command and send this one to the `AppUi`. This class will handle it and quit the application.

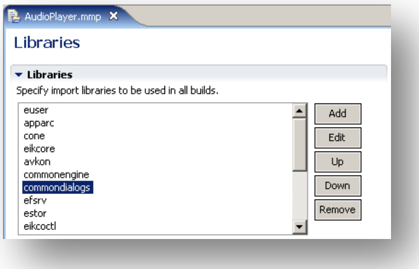
```
TBool CAudioPlayerContainerView::HandleExitMenuItemSelectedL( TInt aCommand )
	{
	AppUi()->HandleCommandL( EEikCmdExit );
	return ETrue;
	}
```

When you start your application again, selecting the *"Exit"* menu item will close the application.

The File Selection Dialog

The next step is to implement the file selection dialog. S60 has got a powerful pre-defined dialog that automatically handles navigating through the file system. However, the UI designer doesn't support creating this one yet, so it has to be done manually.

The memory selection dialog requires linking our application to an additional library. To add it, open the `AudioPlayer.mmp` file (in the `/group/`-folder of your project) to get to the *mmp file editor*. Go to the *"Libraries"*-tab and add the `commondialogs` library:



Specify import libraries to be used in all builds.

euser
apparc
cone
eikcore
avkon
commonengine
commondialogs
efsrv
estor
eikcoctl

The dialog itself has to be defined through a struct in the resource file. Using the `MEMORYSELECTIONDIALOG`-struct for the dialog requires including the appropriate header file. The dialog should be defined in the `AudioPlayerContainer.rssi` (in the `/data/`-folder) file, as it contains the contents that are displayed on the main screen. This is just a convention of the code generated by *Carbide.c++*, you could also put all the resource code into a single main `AudioPlayer.rss`-file.

However, it's best if you add the following line at the top of the `AudioPlayer.rss`-file, as the `AudioPlayerContainer.rssi`-file is simply included into the main `.rss`-file:

```
#include <CommonDialogs.rh>
```

To add the dialog to the container, open `AudioPlayerContainer.rssi` and add the following definition at the end of the file:

```
RESOURCE MEMORYSELECTIONDIALOG r_file_select_dialog
	{
	locations =
		{
		LOCATION { root_path = "C:\\Data\\Sounds\\"; },
		LOCATION { root_path = "E:\\"; }
		};
	}
```

The two locations specified by this resource definition are the two base directories where the user can start browsing through the file system. In this case, it only allows him to view the `\Data\Sounds\`-directory on the internal memory drive `C`, however, he is allowed to browse through the whole memory card.

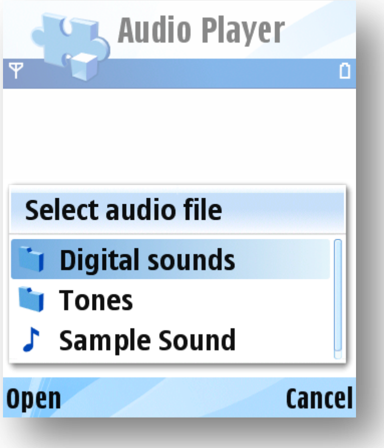
Displaying the dialog is simple. Go to the UI designer and let it create the code to handle the *"Play file"* menu item. The following source code snippet shows what the engine should look like to display the file selection dialog:

```
TBool CAudioPlayerContainerView::HandlePlay_fileMenuItemSelectedL( TInt aCommand )
	{
	// Display the file selection dialog
	TFileName filename(KNullDesC);
	_LIT(KTitle, "Select audio file");
	if (AknCommonDialogs::RunSelectDlgLD(filename, R_FILE_SELECT_DIALOG, KTitle))
		{
		}
	return ETrue;
	}
```

Using this method requires including the following header file in `AudioPlayerContainerView.cpp` or `.h`:

```
#include <akncommondialogs.h>
```

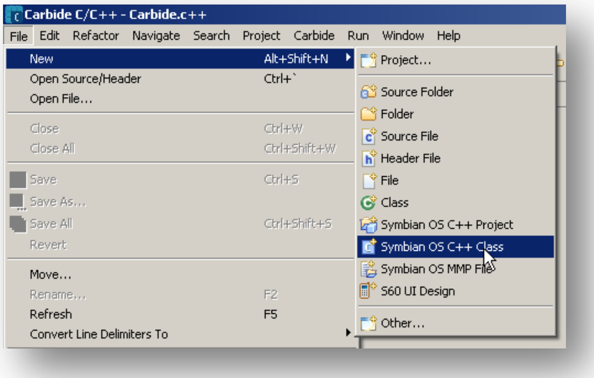
The drives of the mobile phone are also available on the emulator and are mapped to specific directories. To get to the subdirectory that we specified in the resource file on drive `C`, put an *.mp3* or *.wav*-file in: `C:\Symbian\9.1\S60_3rd_MR\Epoc32\winscw\c\Data\Sounds`



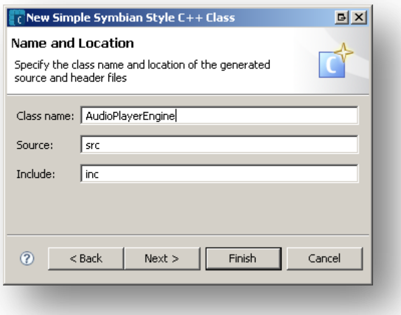
The Audio Player Engine

Defining the Engine Class

The interaction with the client-side of the multimedia server should be handled by a dedicated class. Therefore, create a new class called `AudioPlayerEngine` (source & header-files). *Carbide.c++* features a very useful wizard that can create both files in one step and will even write the two phase construction code for you. Simply go to *File* → *New* → *Symbian OS C++ Class*



In the following dialog, select the *"Simple Symbian Style C++ Class"* template. Make sure that you only enter *"AudioPlayerEngine"* for the *"Class name"*. The wizard will automatically add the `C` to the class name – because the engine is going to be a C type class, derived from `CBase`.



Name and Location

Specify the class name and location of the generated source and header files

Class name: AudioPlayerEngine

Source: src

Include: inc

Defining the Framework of the Engine

Next, we will prepare the general interface and framework of the audio player engine, so that we can add the real communication in the next step – where others can join in, who started with the pre-written framework.

If you think about the tasks of the (minimal) audio engine, it should have the following features:

1. Allow to start playing a file.
2. Allow to stop the currently playing file.
3. Have an own status variable, to make sure that the engine always knows its current state.
4. Allow to query the current state of the engine.

Let's add the required code to meet those requirements. First, define an `enum` called `TAudioState`. It's used to keep track of the current state of the engine. Add this code above the class definition in `AudioPlayerEngine.h`:

```
/** Engine status */
enum TAudioState
        {
        /** No operation */
        EIdle,
        /** Preparing to play a file*/
        EPlayFilePrepare,
        /** Playing a file */
        EPlayFilePlaying
        };
```

Next, we need three new functions, which should be called from the outside to start / stop playing and to query the current state. To start playing, the application should only provide the filename to the engine, which takes care of the rest. Also, a private instance variable for keeping track of the current internal state is defined using the new `enum`.

```
public:
        void PlayFileL(const TDesC &aFile);
        void StopPlay();
        TBool IsPlaying();
private:
        TAudioState iState;
```

Next, it's time to implement those functions. To be on the safe side, it'd be a good idea to make sure that `iState` is initialized with `EIdle`. You can do this directly through the C++ constructor.

The initial implementation of the new methods is printed on the next page. The only task that is important for now is that the state variable is handled correctly. Note that the request to play a file doesn't directly switch the state to the play state, as the player has to be initialized first – more on that later.

```
void CAudioPlayerEngine::PlayFileL(const TDesC &aFile)
	{
	if (iState == EIdle)
		{
		iState = EPlayFilePrepare;
		}
	}

void CAudioPlayerEngine::StopPlay()
	{
	iState = EIdle;
	}

TBool CAudioPlayerEngine::IsPlaying()
	{
	return (iState != EIdle);
	}
```

Connecting the Menu to the Engine

The last part of the framework is to connect the menu / the application to the engine. Additionally, only the menu items which are logically possible at the moment should be visible to the user.

First, go to the UI design of the container again and let it create the methods for handling the ’selected’ event of the two remaining menu items (*“Play last file”*, *“Stop”*).

To let the view use the engine, add the required include statement to the header file of the `CAudioPlayerContainerView`-class, then add a private instance variable:

```
	CAudioPlayerEngine* iAudioEngine;
```

As the engine is an essential part of the view, the instance of it should be created in `CAudioPlayerContainerView::ConstructL()` – use the two phase construction accordingly for this type of variable: consider if you have to use the cleanup stack of not! Don’t forget to delete the object again in the destructor of the view class.

In the menu item handler methods, we will now add the calls to the engine, in order to prepare everything for playing the file. This includes adapting the *“Play”* menu item handler method, so that the `if`-statement is extended to:

```
	if ( AknCommonDialogs::RunSelectDlgLD (filename, R_FILE_SELECT_DIALOG,
KTitle))
		{
		// User selected a file, send it to the audio engine
		iAudioEngine->PlayFileL (filename);
		}
```

Also, call the `StopPlay()`-function of the engine in the *“Stop”* menu item handler method of the view class.

The last preparative step before implementing the actual communication with the audio player is to dim the menu items, which are currently logically impossible. This is done every time the *“Options”*-menu is opened and can be implemented by overriding a method called `DynInitMenuPaneL()` in the view class. It is defined in the base class that our class is derived from (→ `CAknView`) and is automatically called by the framework every time the menu is opened by the user. It allows the application to dynamically adapt the contents of the menu, before it is shown to the user.

Define the method in a private part of the `CAudioPlayerContainerView` class definition:

```
void DynInitMenuPaneL(TInt aResourceId, CEikMenuPane* aMenuPane);
```

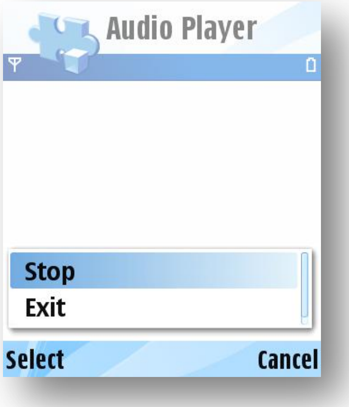
The implementation of this method:

```
void CAudioPlayerContainerView::DynInitMenuPaneL(TInt aResourceId, CEikMenuPane*
aMenuPane)
	{
	if ( aResourceId == R_AUDIO_PLAYER_CONTAINER_MENU_PANE1_MENU_PANE)
		{
		if ( iAudioEngine->IsPlaying ())
			{
			// We're playing, so don't display the play commands
			aMenuPane->SetItemDimmed
(EAudioPlayerContainerViewPlay_fileMenuItemCommand, ETrue);
			aMenuPane->SetItemDimmed
(EAudioPlayerContainerViewPlay_last_fileMenuItemCommand, ETrue);
			}
		else
			{
			// We're not playing, so don't display stop command.
			aMenuPane->SetItemDimmed
(EAudioPlayerContainerViewStopMenuItemCommand, ETrue);
			}
		}
	}
```

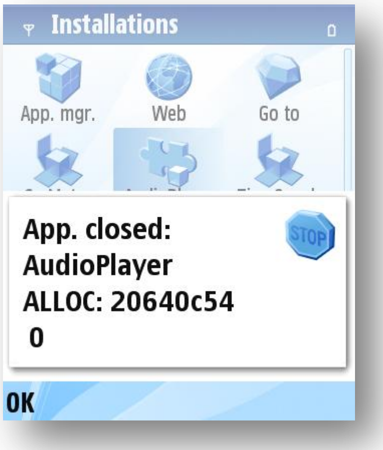
If you followed the tutorial closely, the resource specified by the code should exist (`R_AUDIO_PLAYER_CONTAINER_MENU_PANE1_MENU_PANE`). If there's a problem when compiling the application, make sure the resource id is an uppercase version of the `MENU_PANE` definition in the `AudioPlayerContainer.rssi`-file. Also, you might have to adapt the IDs of the menu items (e.g. `EAudioPlayerContainerViewPlay_fileMenuItemCommand`) if you didn't choose the same labels. You can find the menu command IDs `MENU_PANE` definition as well; alternatively you can take a look at the `AudioPlayerContainer.hrh`-file.

The `DynInitMenuPaneL()`-method essentially checks if the engine is currently playing a file. If it is, it dims both of the "start playing" menu items. If the engine is currently idle, it doesn't allow stopping. If a menu item isn't explicitly set to "dimmed" either in the resource definition or this function, it is visible.

After you select a file in the dialog, the menu should look like this:

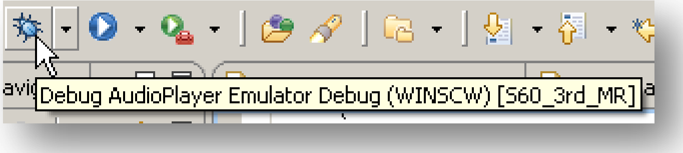


Even though our engine doesn't currently play a file, the menu already adapts to the (theoretical) status of the engine. Remember: Make sure that you quit the application through the menu or the right softkey instead of closing the emulator window. If you get an error message like shown in the following screenshot, you have most likely forgotten to delete the instance of the engine in the view destructor.



```
App. closed:
AudioPlayer
ALLOC: 20640c54
 0
```

The same is true if you get the Windows error message which states that "epoc.exe has detected a problem and it must be closed…". To prevent this crash and see the real error message instead, start the emulator with a debug run configuration instead of using the *play*-button.



Debug AudioPlayer Emulator Debug (WINSCW) [S60_3rd_MR]

This concludes the part of creating the framework. The following sections of this document will have to be executed even if you start with the pre-written framework and skipped the UI design part.

Part 2: The Client Side of the Multimedia Framework

If you start with the pre-written framework, follow the marked edits. This document contains a short description of the individual tasks. You can also use these instructions if you've created the framework yourself and therefore don't have the pre-marked edit positions.

Prepare the Engine for the Audio Player

The first steps include adding the required audio player utility from S60 to the engine. This class handles the client-server communication to the multimedia framework and works in conjunction with the `MMdaAudioPlayerCallback` interface.

Generally, your application sends requests to the asynchronous service provider (e.g. to initialize an audio file, to start playing, etc.) and gets a call-back when this process is finished. Usually you'd have to implement the active object required to handle this yourself – however, for several client-server frameworks, this has already been implemented, thus making it easier for the developer. You just have to implement the call-back interface.

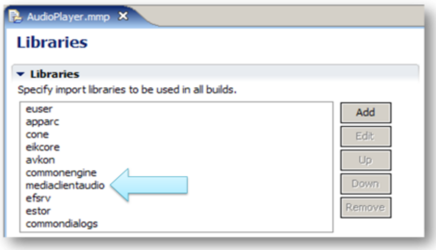
Follow these steps to complete the first task. The numbers correspond to the edits if you use the pre-written framework.

Location: `AudioPlayerEngine.h`

1. Create a new private instance variable for the engine:
   `CMdaAudioPlayerUtility* iAudioPlayer;`

2. This requires including a header file – see the SDK help for `CMdaAudioPlayerUtility`

3. Add the library `mediaclientaudio` to the *mmp file editor*

   Libraries

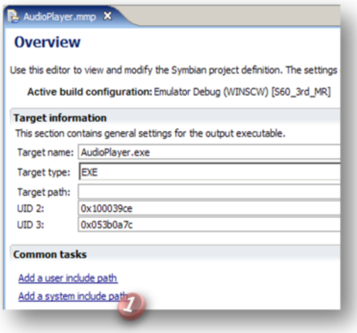
   Specify import libraries to be used in all builds.

   euser
   apparc
   cone
   eikcore
   avkon
   commonengine
   mediaclientaudio
   efsrv
   estor
   commondialogs

4. Additionally derive the engine from `MMdaAudioPlayerCallback`

5. This implies implementing two methods from the callback interface – see the SDK help for `MMdaAudioPlayerCallback`

6. To avoid the following error when working with the S60 3rd Edition MR SDK:
   ```
   WARNING: Can't find following headers in System Include Path
    <mmfPluginInterfaceUIDs.hrh>
   ```
   Add the `/include/mmf/plugin`-directory of the S60 SDK to the *system include paths* through the *mmp editor*:



AudioPlayer.mmp

Overview

Use this editor to view and modify the Symbian project definition. The settings

Active build configuration: Emulator Debug (WINSCW) [S60_3rd_MR]

Target information

This section contains general settings for the output executable.

Target name: AudioPlayer.exe

Target type: EXE

Target path:

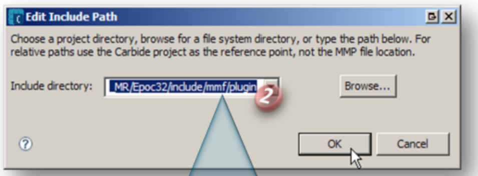
UID 2: 0x100039ce

UID 3: 0x053b0a7c

Common tasks

Add a user include path

Add a system include path



Edit Include Path

Choose a project directory, browse for a file system directory, or type the path below. For relative paths use the Carbide project as the reference point, not the MMP file location.

Include directory: MR/Epoc32/include/mmf/plugin

Find the /include/mmf/plugin-directory within your SDK installation

Communicating with the Audio Player Utility

Location: `AudioPlayerEngine.h` / `.cpp`

7. Initializing the audio player utility is easy – in `CAudioPlayerEngine::PlayFileL()`, create an instance of the `iAudioPlayer` member variable.

   This can be done by using `CMdaAudioPlayerUtility::NewFilePlayerL()`, which supports passing a descriptor (parameter of `PlayFileL()`) plus a pointer to a class that implements the call-back interface – in this case, a `this`-pointer, as the engine implements the `MMdaAudioPlayerCallback` interface.

   Creating the file player automatically causes the multimedia framework to check if it can find a plug-in that can handle the specified file type to initializes the player. When this process is done (successfully or with an error), the `MapcInitComplete()`-method of the engine class is executed.

8. First, let's consider if something doesn't work. In our case, we will cancel playing the file and do cleanup in case of an error. In a real application, the user should of course be notified about the problem.

   As the cleanup has to be done in several occasions, it's best to put this code into an extra function. Define and implement a method called `CAudioPlayerEngine::Cleanup()` and use it to do the following tasks:

   a. If `iAudioPlayer` exists,
   b. Make sure the playback is stopped by calling `Stop()`
   c. Close the connection through `Close()`
   d. Delete the instance of `iAudioPlayer`
   e. Set the pointer to `NULL`
   f. In any case – set the status to `EIdle`

9. The first thing that should be done in `MapcInitComplete()` is to check if an error occurred. The method has two parameters; the first is set to `KErrNone` if everything was ok. If the variable contains a different value, call the `Cleanup()` method and instantly return from the `MapcInitComplete()` method – thus cancelling playback and ignoring the error.

10. If no error occurred, the audio file was recognized and initialization was successful. In this case, set the internal status of the engine to `EPlayFilePlaying`.

11. Next, start playback of the prepared file by calling the `Play()`-function of the audio player utility.

With these changes, playback of the file will already work. Of course, we also have to consider how to cancel playback and how to do proper cleanup after the file has finished playing. These tasks are covered in the next section.

Stopping Playback and Cleanup

Location: `AudioPlayerEngine.cpp`

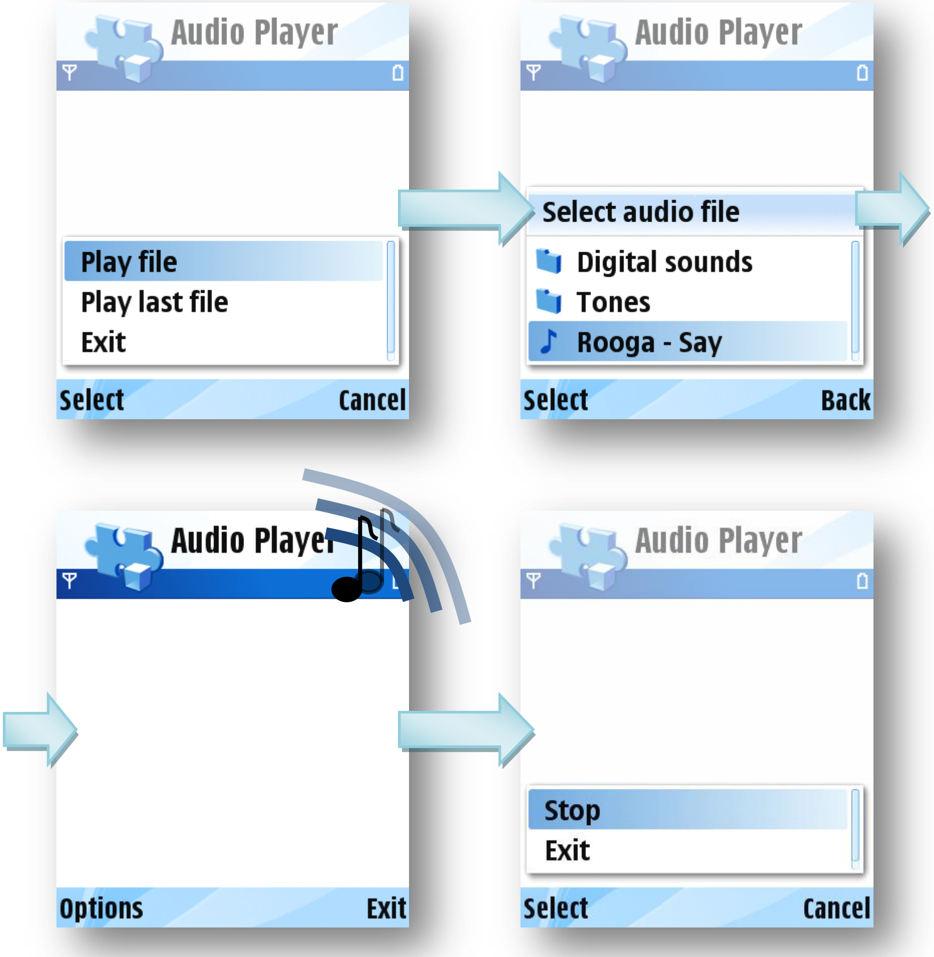
12. When playback is finished, the `MapcPlayComplete()` method of the engine class is executed. The `aError` argument could contain the error code of any problems that occurred during playback, e.g. underflow. For this exercise, the error can be ignored.

    What should be done however is to call our `Cleanup()` method to simply free all the resources. Of course, it would be possible and probably more efficient to reuse the audio player utility class, but it’s easier this way.

13. Should the user exit the application during playback, cleanup has to be performed as well. Therefore, execute the `Cleanup()` method from the destructor as well.

14. The last position where the `Cleanup()` method has to be called is the `CAudioPlayerEngine::StopPlay()` method. This is executed from the view when the *“Stop”* menu item is selected.

This concludes the second part of this exercise. Your application is now able to play any audio files that are supported by the mobile phone – without you having to worry about any codecs or recognizing the file type. The following screenshots show how the application should look like when playing an *mp3*-file stored for the emulator in:
`C:\Symbian\9.1\S60_3rd_MR\Epoc32\winscw\c\Data\Sounds\Rooga - Say.mp3`


Audio Player
Play file
Play last file
Exit
Select
Cancel
Audio Player
Select audio file
Digital sounds
Tones
Rooga - Say
Select
Back
Audio Player
Options
Exit
Audio Player
Stop
Exit
Select
Cancel

Part 3: The File API

Interacting with the S60 Multimedia Framework is one of the “real-life” client-server examples. Thanks to the good implementation of the API, most of the tasks are already pre-implemented and for standard use of the audio player utility, you just have to implement the call-back interface – instead of directly using the client/server interface APIs and implementing an active object yourself.

In this part of the exercise, we will extend the application to persistently save the file that the user played last time. This task is demonstrating how to use the file API of Symbian OS and is a bit closer to the real interaction that is necessary for client/server communication.

Writing the Data File

In the following steps, you will create a method that can save a file containing any data in a descriptor to the private directory of your application. In this case, the data is going be the filename, which is externalized to a file write stream.

Location: `AudioPlayerContainerView.h` / `.cpp`

15. Define and implement the following method:
    `void CAudioPlayerContainerView::SaveDataFileL(const TDesC& aFilename, const TDesC& aData)`

16. In this method, first create a session to the file server and connect to it. It’s recommended to encapsulate the connect statement into a `User::LeaveIfError()` statement, to turn any error code that might be sent out into a leave. If `KErrNone` is returned, no leave will be thrown.

17. Next, push the session to the cleanup stack. Make sure that you use the appropriate function to do so, as the session has to be closed in case of a leave and not deleted!

18. Use the `CreatePrivatePath(EDriveC)`–method of the file server session to make sure the private path of the application exists on drive `C`. You can ignore the return value of this method, e.g. if the path does already exist. If there is a problem with the directory, creating the file will fail in step 20.

19. Set the session path of the file server session to the private directory on drive `C`. Use the `SetSessionToPrivate()`-method.

20. Now, create a file object.

21. Use the `Replace` method to either create a new file or to replace the previous one if it does already exist. Use the `aFilename` parameter and open the file in write-only mode.

22. Push the file object on the cleanup stack. Make sure that you use the correct function.

23. Create a file write stream based on the file object. Look up the required header file for the `RFileWriteStream`–class in the SDK documentation.

24. Push the file write stream on the cleanup stack (R type class, use the `PushL()` method of the file write stream class)

25. Externalize the data to the stream. Note that the externalize operator can leave, so it was indeed necessary to push all the objects on the cleanup stack – as they're all local variables and no instance variables.

26. Call the commit method of the file write stream to ensure that its contents are written to the file.

27. Pop and destroy all three objects using the cleanup stack.

28. Now that the `SavaDataFileL()` method is finished, call it from the `CAudioPlayerContainerView::HandlePlay_fileMenuItemSelectedL()` method. Use a literal called `KDataFilename` for the filename. If you did not start with the pre-written framework, define it to contain a filename like `LastTrack.txt`.

It's time to test if the save-method already works! Start the emulator and select a file to play. Make sure that you close the application in the emulator before closing the emulator window, so that you are notified of any memory leaks your application may suffer from.

If everything went well, the file should have been created. You can find it in the directory `C:\Symbian\9.1\S60_3rd_MR\Epoc32\winscw\c\private\e21a1dd3`, where the last part corresponds to the UID3 of your application – it will most likely be different if you didn't use the pre-written framework.

The data file should contain text like:

```
xC:\Data\Sounds\Rooga - Say.mp3
```

The first part is a representation of the length of the descriptor, which is externalized automatically.

Reading the Data File

Comparable to what we did for writing the data file, we will now write an inverse generic method that can read a file and returns the contents of the file in a heap based buffer descriptor.

Location: `AudioPlayerContainerView.h` / `.cpp`

29. Define and implement the following method:

    ```
    HBufC* CAudioPlayerContainerView::ReadFileLC(const TDesC& aFilename)
    ```

30. Like in the save method, create a file server session object and establish a connection. Then, push the file server session onto the cleanup stack.

    Note: In a real application, you shouldn't connect to the file server every time you need to access a file, as this incurs quite a lot of overhead. Instead, you should save the session as an instance variable and reuse it as often as possible – or even use a file server session that was created automatically by the UI framework.

31. Set the session path of the file server session to the private directory on drive `C`.

32. Define an `RFile` object and open the file specified by the `aFilename` parameter. Use a read only mode.

33. Push the file object on the cleanup stack.

34. Define a `TInt` variable that will contain the file size (e.g. `fileSize`). Use the `Size()` method of the file object to store the file size into the `TInt` variable and leave if the return code indicates an error. This size will be used to create the heap based buffer, which will store the contents.

35. Create a file read stream based on the file object and push it on the cleanup stack.

36. Allocate an `HBufC*` descriptor based on the input stream, with the `fileSize` as the maximum size. Use the `NewL()` method of the `HBufC` class, which can directly accept the stream as the first parameter and the maximum size as the second.

    Note that we don't push the descriptor on the cleanup stack yet, as we have to remove the three other classes from the cleanup stack before we can add the `HBufC` - which should remain there when returning from this method. After all, it's a stack and we can't leave the `HBufC` on the top of the stack and remove the three objects beneath it.

37. Pop and destroy the three file-related objects from the cleanup stack.

38. Push the descriptor on the cleanup stack and return it.

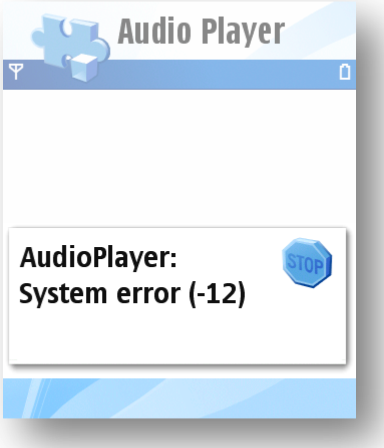
39. Go to the method that is executed when the *"Play last file"* menu item is executed (e.g. `CAudioPlayerContainerView::HandlePlay_last_fileMenuItemSelectedL()`).
    First, read the last filename into a new `HBufC*` variable using the `ReadFileLC()` method. Use the same filename as for the save function, e.g. `KDataFilename`.

40. Now, send the descriptor to the `PlayFileL()` method of the audio engine.
    Hint: Use `*` to dereference the `HBufC*` when passing it to the method as a parameter.

41. Pop and destroy the descriptor from the cleanup stack.
    Remember that the read method left it on the stack, as indicated by the trailing `C` of the function name.

You're finished with this exercise! When you select *"Play last file"* from the menu of the audio player application, it will load the saved file name from the data file and automatically start playing it.

You can also test what happens if you try to use this menu command when no folder/file has been created yet. We did not `TRAP` the leave, so it gets forwarded to the UI framework, which automatically displays an error message – the application itself is not closed down. The reported error code is `-12` (`KErrPathNotFound`) when no folder has been created yet or `-1` (`KErrNotFound`) when the file is missing.



AudioPlayer:
System error (-12)

Make sure that you can still exit your application without any memory leaks, even if there is a leave! If your application passes this test, you have just written your first fully-featured audio player – with just very few lines of code!