

Using an indirection DLL for multi-UI platform support

Andy Weinstein, Degel Software Ltd.
Revision 1.1, September 2005

1. Introduction

An application that supports multiple UI platforms sometimes runs into the need to use different APIs to give similar functionality on different Symbian OS phones. For example, the method for directly sending an SMS message from an application, without user intervention, changed from Series 60 Version 1 to Series 60 Version 2. The old method is not only clumsier to code, but it also simply doesn't work on the newer devices. The new method cannot be run at all on the older devices, because it uses APIs which don't exist in the set of DLLs built into those phones. Another example is *skins* support, introduced in Series 60 Version 2. An application which uses the skins support API will not be able to run on Series 60 Version 1 devices.

A final example is the new *scalable UI* support introduced in Series 60 Version 2 Feature Pack 3. An application written to use these new APIs will only compile on the latest SDK, and can't run on earlier devices. In the attached sample code, we have taken the scalable UI sample application supplied by Nokia and shown how to adapt it so that it can run optimally on both older and newer devices, without taking a "lowest common denominator" approach.

2. Two approaches

In general, a developer faced with these kinds of situation can choose between two approaches. The first approach is to have different versions of the application for each target platform. This is undesirable for a number of reasons, including the need to support multiple code bases, and the need to distribute multiple SIS files to users who must work out the correct one for their phone (which they cannot always do to as Symbian OS phone users become less technical). Even if the code base is managed using conditional inclusion (`#ifdef`), it can quickly become unwieldy (you may want to specify product Ids instead of machine Ids – see [FAQ-1114](#)). Multiple SIS files *can* be bundled inside a single wrapper SIS file with install time directives selecting the proper SIS file for the device, but the wrapper SIS will be a multiple of the size of the actual installable SIS. For a large application, or one which needs to support many different targets, this can result in an unacceptable Over The Air (OTA) download time.

The second approach, presented here, is to use an *indirection DLL* to isolate just those pieces of the application that use platform-dependent API support. These pieces are moved into a DLL as a separate component of the application. This DLL provides a platform-neutral API for the functionality in question. It alone is reimplemented for each target platform, internally using the appropriate platform-specific APIs. All of the different implementations of the DLL are included in the SIS file.

The application is written to call the platform-neutral API in the indirection DLL. The appropriate DLL is selected at install-time, based on the target device. The advantages of this method are that the base source code is shared unchanged across platforms, making support and maintenance simpler, and that the size of the resulting SIS file is not much larger than the original application size. Instead, the SIS file grows only in proportion to the size of the indirection DLL.

3. How to do it

We now describe how to cleanly implement an indirection DLL by using insulation techniques to build an 'Adapter'. Sample code is included which implements such a DLL for a Nokia example application. The Nokia example was written for Series 60 Version 2 Feature Pack 3. The sample code can also run on

previous releases of Series 60 Version 2. We will refer to the example code while detailing the steps to be followed when building such a DLL.

- 1) Define a platform-independent API for the desired functionality. Create a subdirectory in your project for the indirection DLL with an appropriate name. *In the example code, this is actually the top-level directory supplied - the “scalableIndirection” directory. It will be added as a subdirectory to the project as described below.*
- 2) For each new class in the API, compose a header file defining its interface. This header file will consist of a class declaration that declares all of the public API functions, together with a single private member which is a pointer to a class with the same name but with the suffix “Internal” added on. This internal version of the class should be forward-declared at the start of the header file. The definitions of the public API functions should be prefixed with `IMPORT_C`. If there is also a need to define a callback or observer class, that should also be defined in this file as a pure virtual class. Note that only this header file should be included by the main application. No file which contains any implementation details should be visible outside the Indirection DLL. *This step is illustrated by the file ScalableIndirection.h in the example, part of which is shown here.*

```

class ScalableIndirectAknLayoutUtilsInternal;
class ScalableIndirectAknLayoutUtils : public CBase
{
public:
    enum TAknLayoutMetrics
    {
    ...
    };
public:
    IMPORT_C static ScalableIndirectAknLayoutUtils* NewL();
    IMPORT_C ~ScalableIndirectAknLayoutUtils();
    IMPORT_C void LayoutMetricsRect(TAknLayoutMetrics, TRect&);
protected:
    ScalableIndirectAknLayoutUtils();
    void ConstructL();
private:
    ScalableIndirectAknLayoutUtilsInternal* iScalableIndirectAknLayoutUtilsInternal;
};

```

- 3) Write the C++ implementation for the API classes. Because we have limited these classes to have only a single member which is a pointer to the forward declared internal variant, all the implementation details are pushed into the internal class. Therefore a single, identical implementation of each API class is used for all the different platforms. In other words, there should be only one copy of this file in the source tree, together with the header file we just described in (2). This implementation file (or files) will be included in each of the different, platform-specific indirection DLL projects.

With such a shell implementation for each class, which is a pointer to the forward declared internal variant, all the implementation details are pushed into the internal class. Therefore a single, identical implementation of each API class is used for all the different platforms. In other words, there should

be only one copy of this file in the source tree, together with the header file we just described in (2). This implementation file (or files) will be included in each of the different, platform-specific Indirection DLL projects. In our example, this is the file *ScalableIndirection.cpp* of the class, and a destructor which will delete that instance. Each of the public APIs, which were prefixed with `IMPORT_C` in the header file, will be prefixed with `EXPORT_C` in the implementation. The sole content of each of these public functions will be to call an identically named public function of the internal class instance. *This step is illustrated by the file *ScalableIndirection.cpp* in the example, extracted here.*

```
void ScalableIndirectAknLayoutUtils::ConstructL()
{
    iScalableIndirectAknLayoutUtilsInternal =
        ScalableIndirectAknLayoutUtilsInternal::NewL();
}

EXPORT_C ScalableIndirectAknLayoutUtils* ScalableIndirectAknLayoutUtils::NewL()
{
    ScalableIndirectAknLayoutUtils* self =
        new(ELeave) ScalableIndirectAknLayoutUtils();

    CleanupStack::PushL(self);
    self->ConstructL();
    CleanupStack::Pop();
    return self;
}

EXPORT_C ScalableIndirectAknLayoutUtils::~ScalableIndirectAknLayoutUtils()
{
    delete iScalableIndirectAknLayoutUtilsInternal;
}

EXPORT_C void ScalableIndirectAknLayoutUtils::LayoutMetricsRect(
    ScalableIndirectAknLayoutUtils::TAknLayoutMetrics aMetric, TRect& aRect)
{
    iScalableIndirectAknLayoutUtilsInternal->LayoutMetricsRect(aMetric, aRect);
}

```

- 4) For each target platform, make a further subdirectory underneath the main indirection DLL subdirectory which will contain the code for the implementation. *In our example, this gives a directory tree as follows:*

```
scalableIndirection
scalableIndirection\scalableIndirectionLegacy
scalableIndirection\scalableIndirectionCurrent

```

Create header files for the internal variants of each of the classes. The public section of these headers should look very similar to the header file of the external class; the only difference should be the suffix `Internal`, and the lack of the keyword `IMPORT_C`. *We extract a portion of the *scalableIndirectionCurrent* variant of the file *scalableIndirectionInternal.h* here.*

```

class ScalableIndirectAknLayoutUtilsInternal : public CBase
{
public:
    static ScalableIndirectAknLayoutUtilsInternal* NewL();
    ~ScalableIndirectAknLayoutUtilsInternal();
    void LayoutMetricsRect(ScalableIndirectAknLayoutUtils::TAknLayoutMetrics, TRect&);
protected:
    ScalableIndirectAknLayoutUtilsInternal();
    void ConstructL();
private:
    CAknAppUi* iAknAppUi;
};

```

As the classes are implemented, additional private functions and members can be added. In the application's main MMP, reference the LIB file for the indirection DLL. It will have the same name as the DLL. Also, if you have put the API's main header file in its own subdirectory as described in step (1), be sure to add that subdirectory to your USERINCLUDE path. Include the H file in all application files which use the functionality. If you are starting from existing code, which was written for the more advanced platform, be sure to change all references to the advanced APIs which you are addressing to use the indirection version of the API. *These changes are illustrated in the Example section below.*

- 5) Now write the C++ implementation for each of the different internal variants, placing this code in the appropriate subdirectory. This code will either invoke appropriate platform-specific APIs or give stub implementations on platforms that don't have any kind of support for the API in question. *See the file `scalableIndirectionInternal.cpp` in each of the subdirectories in the example.*

```

ScalableIndirectAknLayoutUtilsInternal::ScalableIndirectAknLayoutUtilsInternal()
{
    iAppUi = (CAknAppUi *) CEikonEnv::Static()->AppUi();
}

void ScalableIndirectAknLayoutUtilsInternal::LayoutMetricsRect(
    ScalableIndirectAknLayoutUtils::TAknLayoutMetrics aMetric, TRect& aRect)
{
    if (aMetric == ScalableIndirectAknLayoutUtils::EMainPane)
    {
        aRect = iAppUi->ClientRect();
    }
}

```

- 6) Write an MMP file for each platform, one for each subdirectory. Use the same name for all of the platforms. It is possible to use the LINKAS statement in the MMP to give each of the DLLs an identical internal name, so they will all link at runtime, while maintaining a distinct external file name for each. For example, if you want to be able to see which DLL was installed on a given device by looking in the file browser on that device, using different external names would make this possible.

All the DLLs should share the same UID statement, which should have as its first component the generic DLL UID (0x1000008d), and as its second a UID that is distinct from the application's UID.

The TARGETTYPE should be specified as DLL. While the project is being developed, the MMP file should contain the keyword EXPORTUNFROZEN, so that a LIB file will be produced, in spite of the fact that the exports have not yet been frozen. We discuss freezing the exports later in step (9).

In the SOURCE statements, be sure that the reference to the implementation of the external class is prefixed by “..\”, so that the compilation will find the common C++ file(s) described in step (3). Don't try adding “..” to the SOURCEPATH statement.

Also remember to make a BLD.INF in the same directory, which references this MMP.

See the file ScalableIndirection.mmp in each of the subdirectories in the example, which also contain the relevant BLD.INF files. Here is ScalableIndirection.mmp.

```
TARGET    ScalableIndirection.dll
TARGETTYPE  dll
UID       0x1000008d 0x101FF8E4

LINKAS    ScalableIndirection.dll

SOURCEPATH .
SOURCE    ..\ScalableIndirection.cpp
SOURCE    ScalableIndirectionInternal.cpp

USERINCLUDE  ...
SYSTEMINCLUDE \Eproc32\include \epoc32\include\libc

LIBRARY cone.lib
LIBRARY euser.lib
LIBRARY eikcore.lib

EXPORTUNFROZEN
```

- 7) In the application's PKG file, put in a conditional statement which checks the target device's MachineUID property and installs the appropriate, platform-specific variant of the DLL. Remember that this statement will have to try to be exhaustive. As a default, you will probably want to install the most advanced variant on unknown devices, as any device which was unknown at the time of writing your application will almost certainly not use an older version of the operating system. An alternate way of doing this is to use the DeviceFamilyRev property, which is much more general. But there have been some problems reported with its usage on certain newer devices – so be sure to test! Also, remember that the SDK path for the different platforms will be different; this is what allows all the DLLs to have the same name but still be differentiable in your PKG file! *See an example of the relevant lines in the example section, below.*
- 8) When building, remember to build each of the platform-specific DLLs separately, using the appropriate SDK. The main application should be built using the SDK for the least advanced platform being targeted. It can only be built after the platform specific DLL has been built in that SDK, otherwise it will not find the LIB file.
- 9) Once you have completed the implementation and feel confident that your API is complete (for now!), you should freeze the exports. This will provide backwards compatibility in the future if you decide to

add new functions to your indirection DLL, by ensuring that the order of the functions in the DLL stays the same every time the DLL is built. Freezing is done as follows. First remove the EXPORTUNFROZEN keyword from each of the MMP files. Now go to one of the subdirectories. Run ABLD FREEZE. Note that you will get errors for any hardware platforms for which you have not compiled. These can be ignored, or see ABLD HELP for more details. This process creates a set of new subdirectories which are parallel to the subdirectory in which you ran ABLD FREEZE. The names of these new subdirectories are BWINS and BMARM. You're done! The build process for all of the different variant DLLs will find the appropriate shared DEF files in those subdirectories.

- 10) If you do extend your DLL and want to publish a new version, simply run the ABLD FREEZE command described in (9), making sure that the DEF files for the previous version are in the BWINS and BMARM subdirectories. ABLD FREEZE will take care to update the DEF files by appending new API functions to the end, maintaining the ordinal positions of the older API functions, and thus ensuring backwards compatibility.

4. Example

As we noted previously, we include example code for an indirection DLL which was built for the Nokia scalable UI example and implements a game of Noughts and Crosses. This example does not come as part of the SDK for Series 60 v2 Feature Pack 3. Instead it is available separately for download from Forum Nokia. Look in the Series 60 section, under Code and Examples, Symbian C++ Code and Examples. It's called "Series 60 Platform 2nd Edition Feature Pack 3: Scalable UI Example v1.1". That package actually contains three variant versions of the example application. The one which this article addresses is the V2_3_BMP variant.

As of the writing of this article, no device supporting Feature Pack 3 was available, and the only SDK available was for the CodeWarrior IDE. It is possible to build using that IDE for the emulator, in which case the "Current" variant of the indirection DLL should be built along with the application.

To make a SIS file, use the Feature Pack 3 SDK and CodeWarrior to build the "Current" variant of the indirection DLL targeted for the Thumb platform. Then do a build first of the "Legacy" variant of the indirection DLL using an earlier SDK, and then build the main application using the same SDK. The lines which need to be added to the package file are shown below. The resulting SIS file can be installed immediately on an existing device, and on any new device which supports Feature Pack 3 as soon as it comes out, without modification.

To modify the Noughts and Crosses application to work with the indirection DLL, do the following: Add the attached ScalableIndirection directory to the project parallel to the other directories. Modify the MMP file as follows - change the USERINCLUDE to be:

```
USERINCLUDE ..\inc ..\scalableIndirection
```

Add the library reference line:

```
LIBRARY ScalableIndirection.lib
```

In the files NoughtsAndCrossesContainer.cpp and NoughtsAndCrossesSettingsListBox.cpp add the include file:

```
#include "scalableIndirection.h"
```

Now replace the code for the function HandleResourceChange with this (change the class name for the Settings List Box):

```
void CNoughtsAndCrossesContainer::HandleResourceChange(TInt aType)
{
    CCoeControl::HandleResourceChange(aType);
}
```

```
// ADDED FOR SCALABLE UI SUPPORT
// *****
if ( aType==KEikDynamicLayoutVariantSwitch )
{
    TRect rect;

    ScalableIndirectAknLayoutUtils* scalableIndirectAknLayoutUtils =
        ScalableIndirectAknLayoutUtils::NewL();
    scalableIndirectAknLayoutUtils>LayoutMetricsRect(
        ScalableIndirectAknLayoutUtils::EMainPane, rect);
    delete scalableIndirectAknLayoutUtils;
    SetRect(rect);
}
}
```

In the package file (NAC.PKG), make the following changes:

First make sure that the path references to the Symbian SDK under which you are building the application are correct. The example as downloaded comes with relative references, which assume that you place the example in the same sub-tree as the SDK. Depending on your build environment, you may have problems with path names being too long, and subsequently may find that you have to move the project to the root of your tree. Or perhaps for other reasons you will place it somewhere else. In either case, check the PKG file and modify the path references accordingly.

Add the vanilla second edition Product ID "0x101f7960".

```
;Supports Series 60 2nd Edition
(0x101f7960), 0, 0, 0, {"Series60ProductID"}
;Supports Series 60 2nd Edition Feature Pack 3
(0x102032BD), 0, 0, 0, {"Series60ProductID"}
```

Add the following lines, being careful to modify the portion of the path which refers to the location of each of the Symbian SDK's accordingly, if your Symbian SDKs are not installed in the default directory.

```
; 6600, 6620, 6630, 6680
IF (MachineUID=0x101fb3dd)OR(MachineUID=0x101f3ee3)OR(MachineUID=0x101fbb55)OR
(MachineUID=0x10200F99)
; install legacy specific files
"c:\Symbian\7.0s\Series60_v20\epoc32\release\thumb\urel\ScalableIndirection.DLL"
-":!\system\apps\nac\ScalableIndirection.dll"
ELSE
; install files for feature pack 3
"c:/Symbian/8.1a/S60_2nd_FP3_B/epoc32\release\thumb\urel\ScalableIndirection.DLL"
-":!\system\apps\nac\ScalableIndirection.dll"
ENDIF
```

That's it. It's very simple, because this example application from Nokia really only implements a small part of the full Scalable UI API. But it's a start. Once you have the structure in place, the rest is easy. Have fun!

5. About the author

Andy Weinstein, andyw@degel.com, is a Senior Applications Analyst at Degel Software Ltd., the “Software SWAT Team” – a consultancy of highly experienced software professionals covering a broad range of technologies, with an emphasis on handheld platforms. For more information, see <http://www.degel.com/>.

[Back to Developer Library](#)

Want to be kept informed of new articles being made available on the Symbian Developer Network?

[Subscribe to the Symbian Community Newsletter](#).

The Symbian Community Newsletter brings you, every month, the latest news and resources for Symbian OS.