

# Modularizing Symbian OS Applications

Binary vs. source level modularization of Symbian OS applications

Hamish Willee

Published by the Symbian Developer Network

Version: 1.1 – June 2008

<b>1 INTRODUCTION</b> .....	<b>2</b>
<b>2 REASONS FOR MODULARIZING APPLICATIONS</b> .....	<b>2</b>
<b>3 WHICH APPROACH IS BEST?</b> .....	<b>2</b>
<b>4 EXAMPLE CODE</b> .....	<b>4</b>
4.1 BUILDING THE EXAMPLE CODE .....	4
4.2 PROJECT STRUCTURE .....	4
4.3 PROJECT FILES .....	5
4.4 SOURCE FILES.....	5
<b>5 CONCLUSION</b> .....	<b>6</b>
<b>6 FURTHER INFORMATION</b> .....	<b>7</b>
6.1 REFERENCES .....	7
6.2 ACKNOWLEDGEMENTS.....	7

## 1 Introduction

Symbian OS applications are often split into separate Engine and GUI components. Symbian has historically implemented its Engines as DLLs that are statically loaded by the GUI at runtime. Consequently this method has been enshrined in the literature as ‘the preferred’ approach [R2][R3][R4][R5].

There are good reasons for separating an application into GUI and Engine<sup>1</sup> (see Section 2). However, the Engine does not need to be a DLL – it’s possible to build the Engine as a static LIB which is incorporated into the GUI EXE at link time. It’s also possible to separate the Engine and GUI code at source level, and use the build system to create a standalone EXE for each target platform.

This paper examines all of these methods, and concludes that the approach adopted by Symbian and the device-creation community may not always be the best method for third-party application developers.

On completion of the paper, readers will understand how to structure an application using all of these methods, and will be able to make a reasoned decision about which is best for their use case.

## 2 Reasons for modularizing applications

Typical reasons for using an Engine/GUI split with binary level modularization include:

- different phone UIs can use the same Engine
- when porting an application, only the user interface needs to be changed
- it enables separate development of Engine and GUI components
- reducing the coupling between the Engine and the UI.

Note that while these are benefits of the Engine/GUI split, they are (or can be made to be) true for both the binary and source level separation. Therefore these are not reasons purely in support of a separation into GUI and separate Engine DLL.

## 3 Which approach is best?

Using either a DLL or LIB for the Engine is often more suitable than build level separation when the:

- Application is to be worked on by more than one developer. While it is possible for engineers to jointly work on a project that is build level separated, it is much easier if the components can be built independently. Even where build level separation is possible, a binary separation usually leads to cleaner interfaces and reduced inter-component dependencies.
- UI and Engine vendor are not the same company. While it is possible to separately develop Engine and GUI components using shared sources, it’s not always possible for the developers of each component to share code (legally).

---

<sup>1</sup> Note that this separation co-exists with the application architecture’s class structure’s enablement of the MVC (Model/View/Controller) architectural pattern.

- Application is to be ported to different UIs or operating systems. This can be ‘managed’ using build level separation, but is much easier to use binary level separation.

Using a DLL for the Engine is recommended if:

- Engines are intended to be shared by more than one client. Using a DLL ensures that there is a single copy of the Engine code in memory which is shared by all clients, and that this copy of the Engine can be upgraded without needing to upgrade the clients. In contrast, using static LIB or build level separation of shared code will result in more copies of the code in memory, and all the dependent applications will need to be upgraded in order to fix bugs.
- Engine code must be dynamically extensible. For example, a developer might want to provide an upgrade package from a ‘basic’ to a ‘professional’ version of an application; doing this with a separate code module may be more practical than enabling upgrades through some other method (e.g., update enable codes). This is not possible if all the code is in the EXE itself (as with build level separation or using static LIB for engine code).
- Engine code may be dynamically loaded, reducing the runtime memory footprint.

Using a LIB for the Engine is recommended if:

- You have a library of shared common code (e.g., ‘About’ dialog boxes, IMEI-reading classes etc.). If you use a DLL for this then it has to be packaged in a separate, and separately Symbian Signed, embeddable SIS file. This is more complex and problematic<sup>2</sup> than having the code in a static LIB, which is simply incorporated in the EXE.
- You want to have a self contained Engine but do not want to be bothered by DLL issues:
  - DLL capability assignment can be non-intuitive [R6]. The static LIB is incorporated in the EXE process and gains the EXE’s capabilities – no capabilities need to be assigned to the LIB.
  - Global data in DLLs [R7] has some associated costs and caveats, particularly on the Emulator. Provided the static LIB is to be incorporated only in an EXE (and not DLLs) then using a static LIB does not have these problems.

Build level separation is simpler. This is because there is no need to maintain and build a separate project and project infrastructure for the Engine. As with LIBs, issues related to having a separate DLL do not need to be considered – e.g., DLL platform security issues [R6], exporting interfaces, binary compatibility, and support for global writable static data [R7]. Total code size is also likely to be a little smaller than when using DLLs, although this is not proportionally significant for most applications. A disadvantage of that approach is that the Engine code will need to be built for every target.

Symbian delivers Engines that are built upon by both licensee platforms and third-parties. Not all third-parties that need the source can license it, and the sorts of Engines supplied by Symbian often need to be shared by multiple clients, e.g., PIM engines like Contacts and Agenda. For these reasons, and for the reasons given above, device-creation engineers typically use DLLs for Engine code.

For after-market applications, the GUI is often the only client of the Engine and there are no dependencies on other vendors. Therefore the reasons to recommend DLLs are not compelling and most third-party application developers should build the Engine as a LIB.

---

<sup>2</sup> Note that Nokia S60 3rd Edition has defects related to installing and uninstalling embedded SIS files (see Symbian [FAQ-1487 'Why has my application's dependency SIS file been removed?'](#))

While build level separation has many of the same benefits, static LIBs are preferred because they make it easier to jointly develop and port code – this outweighs any minor gains resulting from a simpler project structure. Build level separation may be considered for small projects, on a single target, developed by just one engineer.

## 4 Example Code

The following sections provide a brief overview of how you might structure an application project using both build and binary modularization. As there is a lot of documentation on how to declare and use a static DLL in the Symbian Developer Library, books, and also the wider Internet, the discussion has deliberately been restricted to the main points.

The example code consists of three simple applications, which add two numbers in response to a menu event, and then display the result. In the first version of the application, the function that adds the numbers is hosted in a separate Engine DLL; in the second, this function is in a separate Engine LIB; and in the third, it's in a separate source/header that is incorporated directly into the UI project. In all cases the source and header files are effectively the same.

Note that the applications are contrived; there is no real need to have a separate Engine for this functionality, and no point at all in having a function to add two numbers, especially when these are hard coded! However, they do serve to illustrate the main differences between projects.

### 4.1 Building the example code

The two applications are provided in the associated ZIP file under `\Modul ar i sati onDLLEngi ne`, `\Modul ar i sati onLI BEngi ne`, and `\Modul ar i sati onSource`.

`\Modul ar i sati onDLLEngi ne` consists of a separate Engine DLL and UI, while `\Modul ar i sati onLI BEngi ne` consists of a separate LIB Engine and UI. These components are built separately from the `\Engi ne\group\` and `\UI QTestApp\group\` folders using standard command line build tool invocation of `bl dmake` and `abl d`. Note that the Engine must be built first.

`\Modul ar i sati onSource` is built from `\UI QTestApp\group\`, again using the standard `bl dmake` and `abl d` toolchain.

### 4.2 Project structure

The applications follow a standard Symbian OS project structure. There are top level folders for each module:

- `\Engi ne` – contains the common Engine code
- `\UI QTestApp` – contains the UIQ-specific application.

The structure is extensible – other top level folders can be added to support additional UI platforms and of course the folders might be split into smaller components.

Within each discrete code module, Symbian typically uses the following project structure:

- `\BWI NS` – contains `. def` files for Emulator environment
- `\EABI` – contains `. def` files for target environment
- `\data` – used for resource and INI files
- `\group` – used for project files
- `\i nc` – header files

- \sis – PKG installation definition files
- \src – source files.

There is no \sis folder for the Engine in any of the projects because in the Engine DLL case it is included as part of the GUI PKG file, and for the LIB and source modularization there is no separate DLL at all. There is also no \data associated with the Engine.

Using source level modularization the Engine structure is much simpler: it requires only the \src and \inc folders. There is no need for the \BWINs and \EABI folders because there is no exported API. There is no need for \group as the Engine is built with the GUI \group, or \sis because there is no separate Engine component. In fact, the headers and source files do not technically even require their own folders – they could be part of the GUI folders. However, having different modules makes it easier to treat the Engine as conceptually separate.

### 4.3 Project files

For DLL and LIB modularization projects there are separate project files for the UI and for the Engine. The Engine BLD.INF file specifies the exported header.

The DLL Engine MMP file specifies the properties of the DLL to be created. The application MMP file specifies the linked Engine DLL in its LIBRARY statement. If the Engine code links against other DLLs, these must also be included in the LIBRARY statement (there are no additional DLLs in this example). The example source code is well documented, and there are other excellent examples in the SDK of how to create and use a static DLL [R1], so is not discussed further.

The LIB Engine MMP file specifies the properties of the LIB to be created. This is simpler than the DLL MMP file – there is no need to define capabilities, UIDs, or LIBRARYs (these must be specified in the EXE that links the LIB). The application MMP file specifies the linked Engine LIB in its STATILIBRARY statement.

For source level modularization there are no Engine project files. The application project files (\UIQTestApp\group\TestApp.mmp) are the same as for the DLL and LIB case except that the MMP file needs to include the source of the Engine directly, as shown:

```
//Engine code locations
SOURCEPATH    .. \.. \Engine\src
USERINCLUDE   .. \.. \Engine\inc
SOURCE        EngineDLL.cpp
```

Note that the build toolchain uses the last-declared SOURCEPATH and USERINCLUDE to locate necessary files, so the Engine SOURCE must follow these declarations.

### 4.4 Source files

The source and header files for all types of modularization are almost exactly the same. In the DLL Engine header file, the prototypes for exported methods are declared with IMPORT\_C, and the implementation is declared with EXPORT\_C, as shown:

```
IMPORT_C static CEngineDLL* NewL();
EXPORT_C CEngineDLL* CEngineDLL::NewL()
{
    CEngineDLL* self = CEngineDLL::NewLC();
    CleanupStack::Pop(self);
    return self;
}
```

The LIB and source projects do not export any methods, so the `IMPORT_C` and `EXPORT_C` have been removed.<sup>3</sup> Similarly, the DLL entry point<sup>4</sup>

```
EXPORT_C TInt E32Dll (TDllReason /*aReason*/)
```

is not required and has also been removed.

## 5 Conclusion

The use of binary level separation is not the best approach for all developer segments.

Developers in the device-creation community will tend to use DLLs because the Engines/Components they create are often intended to be shared by other developers. Using a DLL allows efficient code re-use both in terms of ROM and RAM, and avoids issues related to source distribution.

Third-party application developers should instead use LIBs (except for code to be shared by multiple clients) because in practise they have most of the same benefits, lack some of the problems, and are simpler to use.

Source level separation is only recommended for simple projects being developed by a single independent engineer.

---

<sup>3</sup> Note that this is not strictly necessary; the macros are ignored in EXE projects anyway.

<sup>4</sup> The DLL entry point is only required for EKA1 based platforms, i.e., Symbian OS v8.1x and earlier.

## 6 Further Information

### 6.1 References

No.	Document Reference
[R1]	Symbian Developer Library: Examples > Symbian OS fundamentals example code > CreateStaticDLL and UseStaticDLL: using a statically linked DLL
[R2]	<a href="http://developer.symbian.com/main/learning/press/books/pdf/Coding_Tips.pdf">developer.symbian.com/main/learning/press/books/pdf/Coding_Tips.pdf</a>
[R3]	<a href="http://developer.symbian.com/main/downloads/papers/portable_dialogs/PortableDialogsv1.1.pdf">developer.symbian.com/main/downloads/papers/portable_dialogs/PortableDialogsv1.1.pdf</a>
[R4]	<a href="http://developer.sonyericsson.com/site/global/techsupport/tipstrickscode/symbian/p_splitting_uiq2_application_ui_application_engine.jsp">developer.sonyericsson.com/site/global/techsupport/tipstrickscode/symbian/p_splitting_uiq2_application_ui_application_engine.jsp</a>
[R5]	<a href="http://wiki.forum.nokia.com/index.php/Application_Design_Tips">wiki.forum.nokia.com/index.php/Application_Design_Tips</a>
[R6]	Platform Security - a Technical Overview ( <a href="http://developer.symbian.com/main/downloads/papers/plat_sec_tech_overview/platform_security_a_technical_overview.pdf">developer.symbian.com/main/downloads/papers/plat_sec_tech_overview/platform_security_a_technical_overview.pdf</a> )
[R7]	Support for Writable Static Data in DLLs ( <a href="http://developer.symbian.com/main/downloads/papers/static_data/SupportForWritableStaticDataInDLLsv2.2.pdf">developer.symbian.com/main/downloads/papers/static_data/SupportForWritableStaticDataInDLLsv2.2.pdf</a> )

### 6.2 Acknowledgements

Many thanks must go to John Forrest and Tanzim Husain (Symbian) for their review and guidance, and to Sander van der Wal (mBrain software) for his comments regarding the usefulness of static libraries.

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.