
Symbian OS: Platform- Independent Engine Development

Version 1.0
May 11, 2004

S
Y
M
B
I
A
N
O
S

Legal Notice

Copyright © 2004 Nokia Corporation. All rights reserved.

Nokia and Nokia Connecting People are registered trademarks of Nokia Corporation. Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. Other product and company names mentioned herein may be trademarks or trade names of their respective owners.

Disclaimer

The information in this document is provided “as is,” with no warranties whatsoever, including any warranty of merchantability, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. Furthermore, information provided in this document is preliminary, and may be changed substantially prior to final release. This document is provided for informational purposes only.

Nokia Corporation disclaims all liability, including liability for infringement of any proprietary rights, relating to implementation of information presented in this document. Nokia Corporation does not warrant or represent that such use will not infringe such rights.

Nokia Corporation retains the right to make changes to this specification at any time, without notice.

License

A license is hereby granted to download and print a copy of this specification for personal use only. No other license to any other intellectual property rights is granted herein.

Contents

1.	Introduction	5
1.1	Scope	5
1.2	Goal.....	5
2.	Symbian OS Reference Designs	6
2.1	Understanding the Symbian Vision of Portability	6
2.2	Developer Platforms.....	7
3.	Fundamentals of Symbian OS Application Porting	9
3.1	About Reference Designs	9
3.2	Challenges of Porting.....	9
3.2.1	Low-level example: building tools.....	10
3.3	Basic Programming Pattern for Separating Engine and GUI.....	10
3.4	Advantages of Writing the Engine as a Separate DLL	11
4.	Programming Practices in Porting Application Engines.....	12
4.1	Source Level Porting.....	12
4.1.1	Copy the engine header and source files.....	12
4.1.2	Using macros.....	13
4.1.3	Using inheritance and delegation	13
4.2	Providing a Static Library	13
4.3	Separating the Engine to a Static DLL.....	13
5.	Creating an Engine Library	15
5.1	Static Library	15
5.1.1	Target type definition.....	16
5.1.2	Source and header example	16
5.2	Static Interface DLL.....	17
5.2.1	Overview.....	17
5.2.2	DLL types and their UID definitions.....	17
5.2.3	Setting up a basic application framework – the DLL client.....	18
5.2.4	Specifying the project file (.mmp –file)	18
5.3	Freezing the Exports	19
5.3.1	Adding new exports	20
6.	Conclusion.....	21
7.	Terms and Abbreviations	22

Change History

May 11, 2004	Version 1.0	Initial document release

1. Introduction

This document presents the main principles of developing Symbian OS platform-independent software engines, and reaffirms the advantages of designing modular applications. With a focus on technical issues, it describes the steps required to split a development project into two separate modules. These modules are the actual, portable software engine and its counterpart, the platform-dependent module, which calls the engine through a predefined interface.

1.1 Scope

Currently, there are a wide variety of Symbian OS phones, which makes portability of software extremely important. Successful applications must be quickly available to a wide range of devices and new users in order to have reasonable value. In other words, “time to market” is tighter than ever.

Not surprisingly, the immense interest, together with the increased number of Symbian OS licensees, developers, and software vendors, has made the task of creating portable solutions more difficult in some respects. However, setting the “problem-centric attitude” aside, from the point of view of challenges, it's all just work being co-worked by numerous participants. Portability is not merely achievable — it can be achieved with strength.

It is perhaps more to the point to ask, how does one estimate the extra effort that goes into the decision to use portable modules? This document has another important element: it is written from the point of view that the core business logic, an application engine itself, should be built to be portable. The portability of the overlying user interface (UI) — which wraps this “engine” for the end user — is not handled here.

1.2 Goal

This document presents fundamentals and a working pattern for software developers seeking architectural solutions for creating portable application engines. It provides detailed, technical-level guidelines and instructions for setting up the build process required to deliver the engine as a separate static DLL.

Porting is handled in a general way that enables this information to be used with almost any platform based on Symbian OS, and thus, any Symbian OS Software Development Kit (SDK).

Those readers looking for a complete list or “compatibility map” of current libraries and APIs cross-referenced between all main developer platforms and providing 100 percent portability should look elsewhere. That kind of data is often valid for only short periods of time and is not stable.

Therefore, the goal of this document is to focus on how to create and use a project framework as the basis for writing portable engines for Symbian OS v6.0 and later.

2. Symbian OS Reference Designs

2.1 Understanding the Symbian Vision of Portability

What is a Symbian OS reference design? What was Symbian's vision of a framework for licensees and developers?

A discussion about the UI layer is relevant here, because the transparency of this layer lays the foundation for developing platform-independent software engines. Symbian OS offers a mechanism where interfaces between front-end and business logic are predefined, and reusability of software components has increased value.

Symbian OS reference designs (RDs) originated from the Symbian decision to split the EIKON UI framework of the earlier EPOC OS releases (EPOC OS releases 1 to 5). (Prior to the release of Symbian OS v7.0, the phrase "Device Family Reference Design" [DFRD] was used to refer to what is currently known as "Symbian OS reference design.") EIKON-based architecture was found to be insufficient for maintaining the portability of future applications.

An enhancement was introduced in Symbian OS v6, where the EIKON framework was split into a basic UI system that Symbian named UIKON, and a clearly defined reference design, for example, Quartz, Crystal, and Pearl. Instead of extending EIKON, each licensee had the possibility of crafting an "X-CON" of its own, such as CKON for Crystal.

The following are a few examples of X-CONs from the early days. From the family of Symbian OS v6 smartphones, the UIQ roughly consists of QiKON and EikStd. The UI layer of the Series 60 Developer Platform consists of AVKON and EikStd.

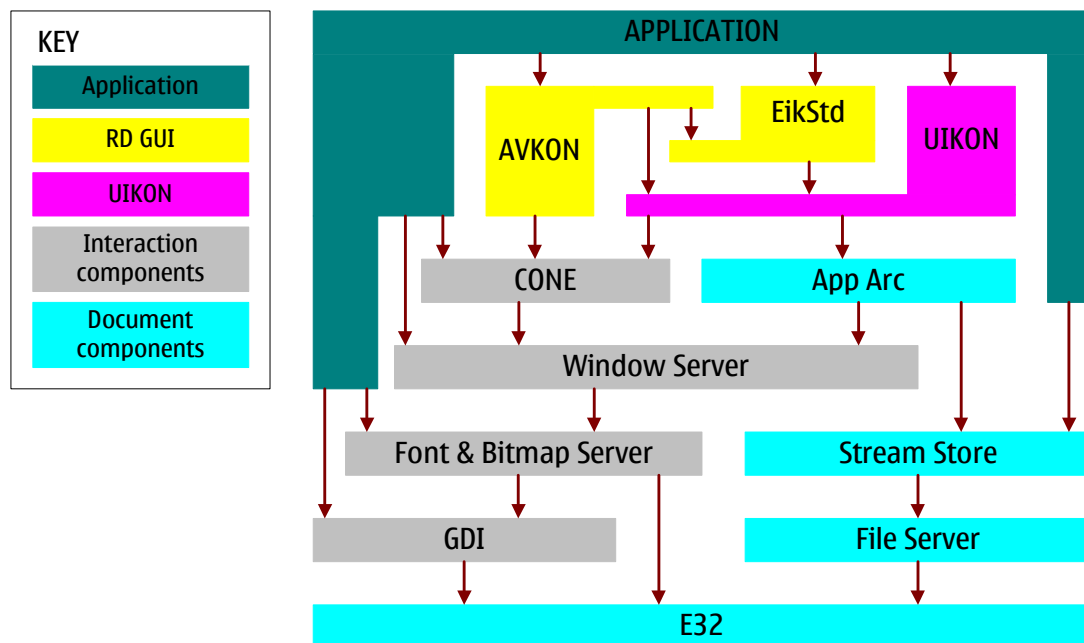


Figure 1: Central APIs of Series 60 Developer Platform

The relationships of these Symbian OS reference design GUIs is illustrated in Figure 1. Both of these X-CONs extend the functionalities of standard UIKON. So, in applications relating these UI class-framework implementations — UIQ and Series 60 Developer

Platform 1.0 — there is customized code on the top of the solution, but the underlying code remains the same.

After the release of Symbian OS v6, licensees started with reference UIs and customized them — keeping the same API along at the same time, with the possible extensions to it. Previously, Symbian used to do this, also producing the UIs, but the new licensing scheme introduced in 2000 led UI owners to take over part of Symbian's role. Today's reference UIs include CKON, QiKON, AVKON, and EIKON.

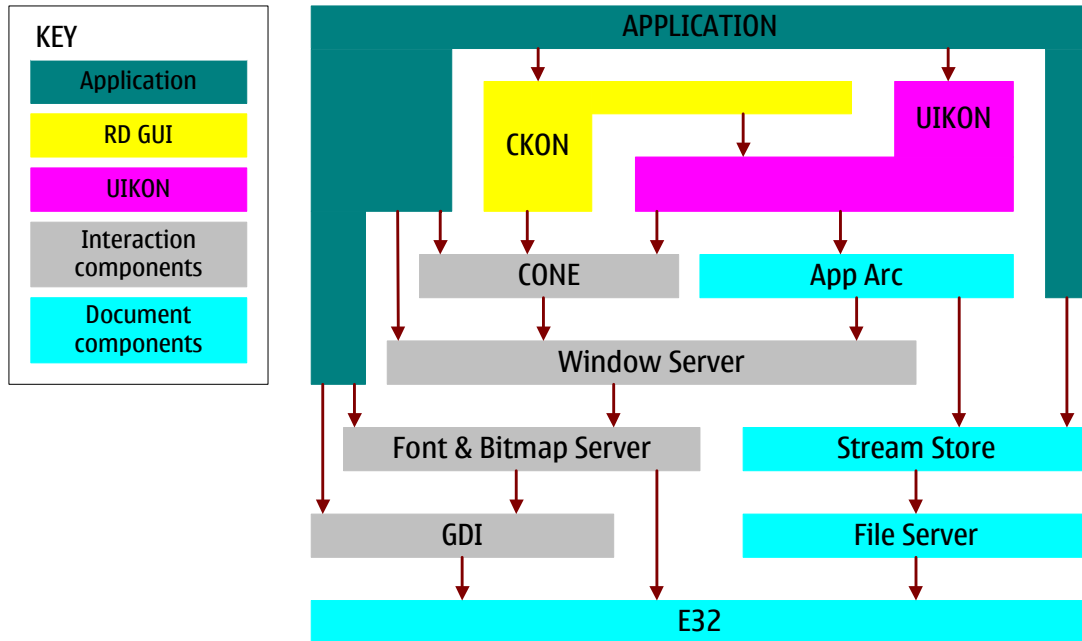


Figure 2: Central APIs of Series 80 Developer Platform

But again, what does this partially UI-centric licensee layer have to do with the porting of application engines? Well, it has at least two major impacts.

First, as the history of the past few years has shown, a great number of new Symbian Platforms have emerged in a short time, many of which having a customized licensee layer on top of the common Symbian OS version. So, for a Symbian software product to reach a reasonable number of end users, the application has to be ported to various devices. Reusable components have a key role in cutting overhead and time-to-market.

Second, the whole reorganization of Symbian OS APIs achieved by Symbian was *intended* to encourage the modularization that platform-independent engines represent and offer at their best.

2.2 Developer Platforms

Today, Symbian OS partners and especially licensees and device manufacturers have a significant role in the creation of the next generation of Symbian OS. This new model offers tailored Symbian platforms for software vendors around the world, and these platforms are often presented as a complete SDK that is a tailored extension of a certain Symbian OS version, for the device family of a certain mobile device manufacturer.

A certain version of Symbian OS forms the basis of each developer platform. Based directly on this is a “licensee layer,” which includes the extensions to the basic functionalities of one operating system release. In the field of this licensee layer are “players” like Nokia and Sony Ericsson, who offer services to other device manufacturers

and, of course, to the application and service developers and independent software vendors targeting their products.

In March 2004, more than 85 percent of the world's mobile phone manufacturers licensed Symbian OS. These manufacturers include Armima, BenQ, Fujitsu, Panasonic, LG Electronics, Motorola, Nokia, Samsung, Sanyo, Sendo, Siemens, and Sony Ericsson.

One of these Symbian OS licensees, Fujitsu, has a development platform whose core is based on the Series 60 Platform that Nokia licenses. The Fujitsu Development Platform consists of all of the preceding layers, offering an add-on to the generic Series 60 Platform.

3. Fundamentals of Symbian OS Application Porting

3.1 About Reference Designs

The current Symbian OS releases and their reference UIs give licensees the possibility of offering developers effective ways of building portable applications. This situation encourages developers to keep business logic separate from the UI implementation. In rough estimates, almost 80 percent of code can be kept intact and unchanged while porting an application — with a good architectural design — when moving from one reference UI to another. This means that only 20 percent of the code would need to be rewritten, as shown in Figure 3.

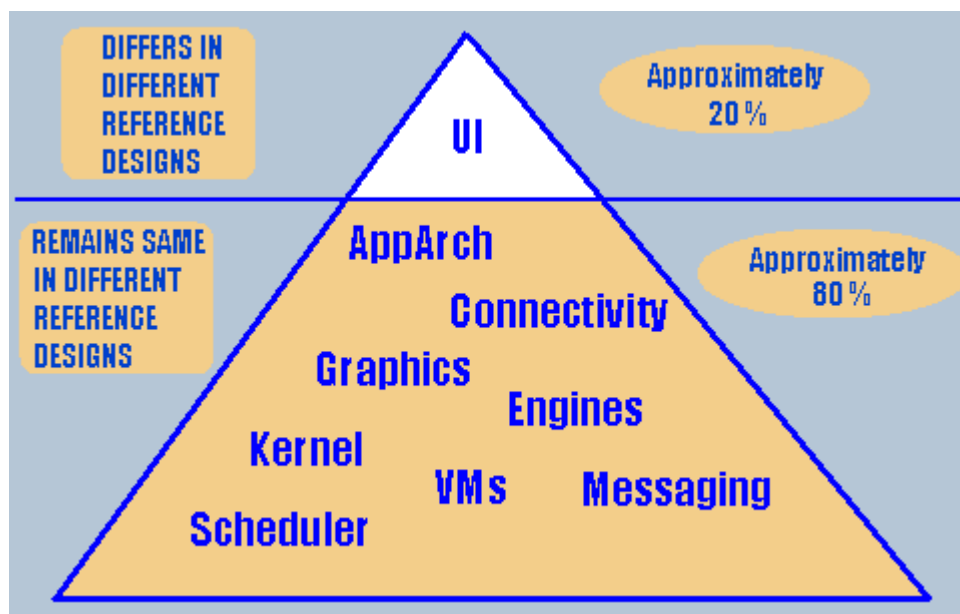


Figure 3: An iceberg graph demonstrating weight ratio between reference UI code and RD independent code

3.2 Challenges of Porting

A great number of applications developed for Wireless Information Devices (WIDs), such as smartphones, are small and GUI-oriented. This hinders developers coding the engine into a separate module behind a strict interface because splitting small-scale work into two separate projects generates extra work. And even though separate engines could be produced, it is not reasonable if the actual amount of portable code is nearly fictitious. This is due to the fact that details about compatibility between developer platform APIs are not always easy to discover, and often research must be conducted before it can be determined whether a certain engine can even be produced.

Another challenge arises from the competitive business environment, which leads licensees to offer some functionalities before they are even rooted to their developer platforms, that is, incorporated into the next version of Symbian OS. This leads to compatibility problems in the future — for example, overlapping implementations from entirely different sources coexist for multimedia message support.

3.2.1 Low-level example: building tools

For new programmers interested in the current type of Symbian OS C++ development, an understanding of the basics of the application framework demands more than traditional C++ programming. Programmers learn a certain design pattern that is well-suited for small-scale projects and prototype applications. Thus, designers need a new orientation toward the ideas of portability.

For example, the Nokia 9210 Communicator SDK contains a tool based on Java™ technology, the Minimal EPOC Application Developer, which can be used to provide separate projects for engine and GUI modules. While the reasons for dropping the tool from the newer SDKs are not plainly as technical as this, it could be read as a possible intention to simplify the basic examples of minimal class frameworks for new developers.

Today's build tools for SDKs for Symbian OS v6.1 and later are capable of producing "work spaces" (also known as "solutions") for the most commonly used Integrated Development Environments (IDEs). However, if multiple projects (Mobile Media Platform [MMP] files) are defined for the build process (bld.inf), each project will have a "work space" of its own, instead of one that includes all of the defined projects (for example, one for DLL and one for GUI).

This build-process-related problem is more a problem of the unification of common make and build tools and IDEs from different vendors, and not related to Symbian development in any way. However, the result is that software designers must manually maintain the workspace where project parts are combined. Alternatively, the coder could keep on working with several projects with the time penalty taken from switching between the projects and IDE views.

3.3 Basic Programming Pattern for Separating Engine and GUI

In practice, it has been effective to split applications into at least two separate modules: the user interface (UI) and the engine. Here, the user interface specifies the method by which the user interacts with the application. The engine defines the functionality of the application itself, *including* the way the program interacts with system devices.

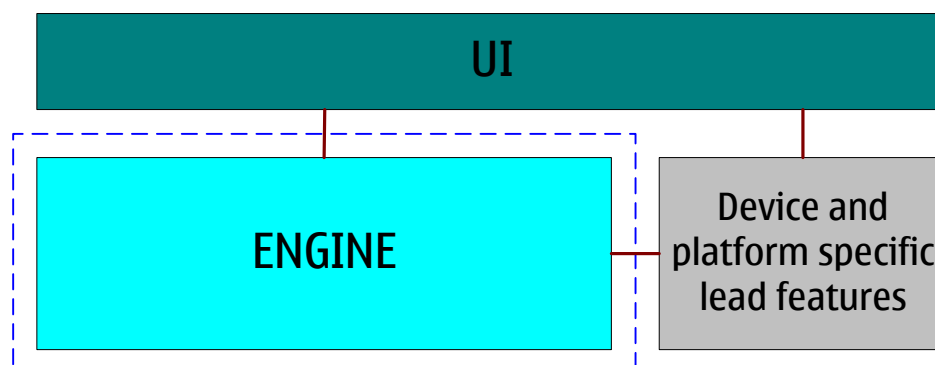


Figure 4: Separate engine and UI

Like any other modern operating system, Symbian OS offers components to assist a programmer in developing the engine and the GUIs. On the technical level, this means a programming practice where the engine and different GUI elements such as views have implementing classes of their own.

What are the advantages of this method? To begin with, small applications are much easier to test. And it also makes it possible to replace the application UI with another

that has a different look and feel, with significantly less work than if no separate engine existed. For example, the first Symbian OS v6 smartphone, the Nokia 7650 device, has the same engines behind the new, modified user interfaces that were first created for the Nokia 9210 Communicator. Only the screen, with its new size and look and feel, has changed.

3.4 Advantages of Writing the Engine as a Separate DLL

Is there an even more advanced and sound solution for taking the engine development completely out of the GUI project? The quick answer is yes. The engine code can be compiled into a separate DLL, making this module a platform-independent, portable object. Actually, this scenario has existed from the beginning of the Symbian Community; we are simply reemphasizing it.

Symbian OS supports applications to load custom-made DLLs; the development of DLLs has also been simplified. Symbian SDKs and their respective build tools enable a way of defining multiple projects into the same build chain. So, it is possible to make a DLL project for the engine and one or more client application projects that use this DLL and interact with the end user (for example, GUI applications).

4. Programming Practices in Porting Application Engines

This chapter discusses the main challenges encountered while trying to create a portable, platform-independent engine. It also introduces methods by which these obstacles can be tackled and solved.

Portability for engine modules can be achieved on three levels:

- Source level
- Library level
- Static DLL level

Each of these approaches has pros and cons that must be fitted to application-specific requirements. The first option, source level porting, is easiest to handle with traditional Symbian development methods, because in this case the engine is a class that should have a strictly defined interface to other classes. This option is easy to create and update, which makes development fast.

The next approach is to create a separate library (which usually has a .lib extension). The library has one or more precompiled objects ready to be linked with the other possibly nonportable modules of each platform, where it is later imported using the corresponding header files for the library.

Both source and library levels require a new build and compilation of the application when the engine is updated. If the same class or library is used by multiple applications in the device, the resources are wasted because the same data is being used without sharing.

The third approach, in which a static DLL is designed, solves some aspects of this problem. Because Symbian OS itself loads the library into memory and makes it accessible to any application, the duplicates are eliminated. At the same time, if the engine is internally improved, the static DLL may be upgraded to devices without touching the applications dependent on its services.

4.1 Source Level Porting

Here is a short summary of methods appropriate for small-scale applications and projects.

4.1.1 Copy the engine header and source files

First, the engine is developed into a separate class, keeping portability in mind. Next, the file is copied to the porting project of a new platform, just as it is, and adapted to fit with the other, nonportable modules and components — such as UI classes — by making modifications to the source.

There is a constant temptation for developers to touch the engine's public interface, because it exists as an accessible source. This may lead to version management problems during development because the source files quickly start to follow separate paths. To keep work overhead to a minimum, developers may use macros, inheritance, and delegation, which all have a common goal — to keep the engine source as untouchable as possible.

4.1.2 Using macros

With macros, the nonportable differences are solved quickly. On the other hand, as the code becomes more complex, a relevant decrease in its readability is inevitable and the whole solution becomes difficult to maintain. However, macros are efficient for small-scale solutions that are in need of a rapid means to be ported to another platform.

Macros are handy in common situations where the target platform has a similar API function, but differs enough to exclude direct porting, for example when one platform's method leaves, and the other doesn't, as in the example below.

```
#ifndef _UNICODE // Signals Symbian OS v6.0 and later
void CMyControl::SizeChanged()
#else // Symbian OS v5 and earlier
void CMyControl::SizeChangedL()
#endif
{
    // Non-leaving implementation
}
```

The above solution is acceptable only if it is possible to have a non-leaving implementation. When used in isolated instances, and not excessively, the macro approach works well.

4.1.3 Using inheritance and delegation

Another possibility is to write the portable engine as a base class, and subclass it to different platforms. This sounds like a neat process, but the reality is that when all of the features that most of the application's engines offer (observer, listeners, etc.) are evaluated, there is no common base class that is available in all main platforms. So, why make a new base class at all? This leads one to reconsider if the copy-and-adapt-files method is enough after all.

4.2 Providing a Static Library

Normally, object files compiled from the source are linked into a single application file (which, in the case of a GUI application, has an APP extension). This polymorphic DLL offers an entry-point function at a known location (a "main" method), which is called by the operating system when the application is executed. However, Symbian build tools provide the possibility to link objects into a lib file, which can be linked to other sources using the library's headers.

As an advantage, the source files need not be given at all, which works for cases where the engine is sold to developers. The libraries might also be used inside a company as an aid for version control: a certain team of software designers may provide the library for others. Configuration management is simplified because the engine module is inside this precompiled library file.

4.3 Separating the Engine to a Static DLL

An easy, efficient way to achieve a portable engine is to write it to a separate static DLL. In this approach, the business engine's code, which possibly uses certain API functions that are compatible between Symbian OS releases and reference designs, is encapsulated behind a static DLL interface.

The DLL offers its own, static set of methods, which are available to all UI classes and, if properly designed to be clear of compatibility conflicts, it is ready to fit in for Reference Design environments. In Symbian OS, static DLL may not contain public, global data

members because of limited memory resources. The reason is simple — if such data members existed, the memory reservation would be multiplied by the number of applications using the DLL. In many cases, the shared data area would seem to be efficient to design, but due to the small memory environment (for example, compared to a PC workstation OS), is impossible with Symbian architecture.

5. Creating an Engine Library

Symbian OS, like other modern operating systems, offers a tailored framework for object-oriented software development. The framework consists of abstract base classes and classes designed for extension. This framework is delivered as part of Symbian OS in system DLLs. These conventional static interface DLLs export functions that may be called by other code being built against system DLL header files and linked against a corresponding DLL import library.

To implement a new application, a programmer usually derives one of the system's abstract base classes and extends it to have new features where required, thus making use of the ready, default base-class-level implementation. Traditionally, new applications are built into polymorphic interface DLLs, which do not require an import library because polymorphic DLLs export only a single function at a predefined location. A call to this function instantiates a newly derived framework class. With GUI applications, this function is `NewApplication()`, which is used to construct such a class.

In addition to creating more advanced static interface DLLs, Symbian OS build tools also provide the possibility of creating plain, static libraries. They are used the way that static libraries are used by older operating systems, where these libraries need to be statically linked with the executable using them.

The Symbian OS build environment has a powerful set of tools, where a complete project is fully specified by a single project file (which has an MMP extension). Because these tools are capable of generating make files using a project file, the target type of a project ensures the correct build process, which produces the appropriate type of target.

The target types discussed are:

- APP — a standard GUI application, a polymorphic interface DLL
- LIB — a static library
- DLL — a static interface DLL

When a software project grows and becomes complex in functionality, it reaches a point where solid, modular architecture quickly starts to pay off. On many occasions, it has been profitable to provide abstraction and modular design by splitting some parts of the project into one or more static interface DLLs, also known as shared library DLLs.

Using of static DLLs in Symbian OS also improves maintainability, because only DLLs may be upgraded into the devices running applications that call DLL code. For example, if a platform-independent, three-dimensional engine is in the form of a static interface DLL and can be uploaded transparently over the network to end users, it is a significantly more elegant solution for software developers and service providers.

5.1 Static Library

Is there any point in making “old-style” static libraries, if they add complexity to development but lack the same level of abstraction as static interface DLLs, and both require an almost equal amount of extra work? The question contains part of the answer: a static library represents a midpoint between a standard application and static DLL projects.

The use of static libraries should be seen as an option in cases where no DLL is required, but modularization and even low-level version control can be accomplished by implementing some part of a project as static DLL. And, when static libraries are used

from the start with a middle- or large-scale project, it is easy to convert the application to later make use of a static DLL framework.

5.1.1 Target type definition

Symbian OS has target type *lib* defining that the compiled object codes should be linked into a static library. As with other target types (for example, APP and DLL), this will set UID1 correctly, and no other UID definitions are needed. Here is a sample MMP file:

```
// StaticLibEngExample.mmp
TARGET          StaticLibEng.lib
TARGETTYPE      lib
SOURCEPATH      ..\src
SOURCE          StaticLibEng.cpp
USERINCLUDE     ..\inc
SYSTEMINCLUDE   \epoc32\include
LIBRARY         euser.lib
```

Most of the properties are similar to those of a standard .MMP file. The **TARGET** line defines the file name of the library. After a successful compilation, the linker will produce a library file that will be stored in an appropriate platform's release directory. For a debugging version of the WINS compilation, the path is <epoc root>\release\wins\udeb.

5.1.2 Source and header example

With a static library, a corresponding header file must be included. To make it possible for other code to link against the library, an import header must be supplied. The Symbian OS framework offers an **IMPORT_C** macro to be used while designing this header, as in the following code example:

```
// StaticLibEng.h
#ifndef __STATICLIBENG_H
#define __STATICLIBENG_H
#include <e32std.h>
class CStaticLibEng
{
public:
    IMPORT_C TPoint GetLoc();
private:
    TPoint iLoc;
};
#endif
```



Note: The library is not derived from a typical CBase class, because it does not make use of the heap, and therefore could have been defined instead as a simple T-class (stack). However, for a reasonable code segment to exist in the library, it often grows over the limitations of a T-type class with stack-only allocated members.

Global, writable static variables should not be defined, because they cannot be used in Symbian OS if the library is later converted into static interface DLL. The static writable data would need to be copied into the run-time memory of each process using the DLL, which would soon lead to a lack of memory because the minimum possible allocation size is 4 KB (in conventional MMUs). Instead, Symbian OS offers a "Run in Place" feature, where code can be run directly from the file system, because this "disk" is actually memory (ROM).

In the source file of the static library, the functions offered by its interface must be exported. This task is a good way to use `EXPORT_C` macro, as in the following example code:

```
// StaticLibEng.cpp
#include "StaticLibEng.h"
EXPORT_C TPointint CStaticLibEng::GetLocFoo()
{
    return LociBar;
}
// requirement for E32 DLLs
EXPORT_C TInt E32Dll(TDllReason)
{
    return 0;
}
```

The `EXPORT_C TInt E32Dll(TDllReason)` function is not mandatory for the static library, but it is required by the Symbian OS framework for every DLL; creation of the latter is discussed in the following section.

5.2 Static Interface DLL

5.2.1 Overview

As stated earlier, for a client program to use the functions provided by a static DLL, it must be compiled against the `.h` file (`#include`) and linked against the DLL's `.lib` file. The `.dll` file contains the executable code that is called at run time.

Under WINS/WINSCW, the C++ source is compiled to object files in the build directory; these are then linked to the release directory. The programmer must run an executable program to invoke this DLL. Other programs can be built to use the DLL, by linking against the `.lib` that was built with the DLL.

Under ARM targets, the C++ source is compiled using GNU C++, and then linked into the release directory. The tools ensure that all DLLs for ARM are linked by ordinal, so as to minimize their size. The programmer may transfer the DLL to a target machine, along with suitable programs to allow it to be executed. The developer should leave the `.lib` in the release directory, in order to be able to build other programs that use this DLL.

5.2.2 DLL types and their UID definitions

There are two other common DLL types in addition to the static interface DLLs supported by Symbian OS.

Table 1 shows a list of DLL subtypes and the source of their corresponding UID definitions. The constants (such as `KSharedLibraryUid`) derive from `e32uid.h`, which is common to all SDKs.

Target Type (description)	UID1 (constant)	UID2 (constant)	UID3
<code>ExeDLL</code> (e.g., a daemon type application, "a service")	Defined by MMP file's target type <code>ExeDLL</code> (<code>KExecutableImageUid</code>)	Not required	Not required

Target Type (description)	UID1 (constant)	UID2 (constant)	UID3
APP (polymorphic DLL, e.g., GUI application)	Defined automatically by MMP file's target type: DLL (KDynamicLibraryUid)	Defined by framework, in GUI applications: KUidApp	Framework dependent. In GUI applications, selected by developer
DLL (static interface DLL, shared library)	KDynamicLibraryUid Defined by MMP file's target type	Must be equal to KSharedLibraryUid, which equals 0x1000008d	Selected by developer

Table 1: The main target types for static interface DLLs in Symbian OS

DLL UIDs allow the loader to check whether the DLL is of the subtype intended. This prevents the inadvertent loading of files that just happen to have the same name.

For polymorphic DLLs, UID2 is defined by the framework being implemented. Some frameworks require a UID3 to be specified, others do not. For example, a GUI application must specify a UID2 of `KUidApp`, which is defined by the application framework, and a UID3, which must be allocated by the developer. To give another example, the printer driver framework requires a UID2 but not a UID3.

Where one project uses multiple DLLs, for example a GUI application comprising a framework-derived polymorphic DLL with an application engine built separately as a static interface DLL, the DLLs will usually share the same UID3. In other cases where multiple projects share some common DLLs, for example multiple different applications that share some engine DLLs, each project, including the shared engine project, will specify unique UID3s.

5.2.3 Setting up a basic application framework – the DLL client

It is easier to describe the static DLL implementation when operating under the assumption that the basic Symbian OS framework classes (constructing the client for the DLL) are available and a complete project file structure with appropriate build tool files is ready.

At a minimum, it assumes the presence of Application and Document classes, and possibly AppUi and View classes for GUI applications. The details are not discussed here, because good examples are available in all Symbian OS development platform SDK documentation, beginning with minimal HelloWorld applications.

In conclusion, a typical Symbian OS framework application should exist, where the target type is APP. The easiest way to produce this in the WINS/WINSCW environment is to use an application wizard capable of producing the required files.

5.2.4 Specifying the project file (.mmp –file)

The developer needs to create a new project file for static interface DLL. The file usually has an .MMP extension; one way to approach it is to arrange for it to be

placed into a project's group directory, where the DLL client's project file exists, too.

First, define the target type in TARGETTYPE line as `dll`. This will force the UID1 to be correct.

Second, specify a UID for the DLL in the UID line, such as:

```
UID 0x1000008d <UID3>
```

UID2 must be `0x1000008d`, and UID3 is to be selected by the developer.

Here is an example of an MMP file:

```
// StaticDllEng.mmp
TARGET StaticDllEng.dll
TARGETTYPE dll
UID 0x1000008d 0x19760111
SOURCEPATH ..\engine
SOURCE StaticDllEng.cpp
USERINCLUDE ..\inc
SYSTEMINCLUDE \epoc32\include
LIBRARY euser.lib
EXPORTUNFROZEN
```

The UID3 will be used by the loader to uniquely identify the static DLL at execution time. The previous example configuration will use the name of `StatiDLLeng[1976011]` inside an ARM-compiled import library. When `StatiDLLeng.dll` is requested by a new client application, the loader checks that its UID is `0x19760111` — and if it isn't, the load fails. If the library is already loaded, it is attached with the new client using it.

The following section will explain the `EXPORTUNFROZEN` line.

5.3 Freezing the Exports

When the DLL interface is ready, it is highly recommended to freeze its exports. This is done to ensure backward compatibility of new releases of a library. Every subsequent change in the DLL's interface that changes its design (changes exported functions) eats up the advantages the DLL was built for. So developers need to pay close attention to the interface architecture right from the start.

During development, the `exportunfrozen` keyword in the project's MMP file can be used to tell the build process that exports are not yet frozen. When the developer is ready to freeze, he or she should remove the `exportunfrozen` keyword and supply a `.def` file listing the exports.

Creating a `.def` file is easy. Just build the project in the normal way and a warning will be displayed because the frozen `.def` file does not exist yet. Once the project is built, the exports can be frozen by calling the freeze target in the makefiles using `abld` batch file.

```
> abld freeze
```

This command produces the frozen `.def` file, which contains the project's exported functions. Note that all ARM platforms share a common `.def` file, but `WINS/WINSCW` has a different `.def` file.



Note: Exports could be frozen in this way even if the `exportunfrozen` statement is specified, but the import library will be created as a side effect of linking rather than from the frozen `.def` file, and this import library will be created whether the project is frozen or not.

Once the project is frozen, regenerate the makefiles so that the import library will be created directly from the frozen .def file. Use the following commands:

```
> abld clean
> abld makefile
```

5.3.1 Adding new exports

New exports can be easily added to the frozen .def file by calling the freeze target in the makefiles once the project has been built with the new exports.

The `abld`'s freeze target calls a tool, `efreeze`, to compare the frozen .def file if one should exist, to the newly generated one by the two-stage link process in the following directory:

```
<EPOC root>\build\<absolute path to MMP file>\<mmp
basename>\<platform>\
```

The `efreeze` checks that the frozen exports are all present and correct in the generated .def file, and appends any new exports to the end of the frozen .def file.

6. Conclusion

For large-scale projects, products, and widely distributed applications that may offer reusable functionality, it is effective to build a software engine into a static interface DLL. Symbian OS and the licensee platforms both offer and direct ways of making these kinds of modular applications.

As a simple rule, a project is small enough to remain as a normal executable as long as it does not offer any new aggregate of functions that could be separated and offered to other solutions. However, when an application reaches the point where its architecture starts to play a bigger role, it often pays to render it to a platform-independent static DLL.

7. Terms and Abbreviations

Term or abbreviation	Meaning
DFRD	Device Family Reference Design. This term has not been in official use since the release of Symbian OS v7.0. Currently, the closest equivalent term is simply “reference design,” or RD.
DLL	Dynamic Link Library.
UI	User Interface.
UID	Unique Identification number. Each released Symbian platform application needs to have at least one UID. To ensure that each UID is genuinely unique, the Symbian Developer Network allocates UIDs from a centrally administrated database.