

# Porting from Windows Mobile to Symbian OS v9.x

**Paul Todd**

**Published by the Symbian Developer Network**

**Version: 1.0 – September 2008**

<b>1</b>	<b>INTRODUCTION .....</b>	<b>2</b>
<b>2</b>	<b>COMPARING MICROSOFT WINDOWS MOBILE AND SYMBIAN OS.....</b>	<b>2</b>
<b>3</b>	<b>GETTING STARTED .....</b>	<b>6</b>
<b>4</b>	<b>PORTING TIPS .....</b>	<b>9</b>
<b>5</b>	<b>RELEASING THE APPLICATION.....</b>	<b>14</b>
<b>6</b>	<b>USEFUL TOOLS.....</b>	<b>15</b>
<b>7</b>	<b>SUMMARY.....</b>	<b>15</b>
<b>8</b>	<b>GLOSSARY .....</b>	<b>16</b>
<b>9</b>	<b>TECHNICAL RESOURCES .....</b>	<b>16</b>
<b>10</b>	<b>AUTHOR PROFILE .....</b>	<b>17</b>

## 1 Introduction

When someone familiar with developing for Microsoft Windows Mobile and the Microsoft Windows platform moves to developing for Symbian OS, there is a period of uncertainty about where to start. The APIs, tools and development environments are so dissimilar that it comes as quite a shock to many.

The intent of this paper is to provide intermediate and advanced Windows CE/Windows Mobile developers with an introduction to porting applications from Windows Mobile to Symbian OS v9.x.

## 2 Comparing Microsoft Windows Mobile and Symbian OS

Windows CE is the base platform for Microsoft mobile technologies. Windows Mobile and Pocket PC<sup>1</sup> are customizations of the Windows CE platform and 90% of the APIs are the same across all the platforms, including Windows desktop and server. Symbian OS is very similar to Microsoft Windows CE in that it is a base platform which is customized by licensees and OEMs to produce new platforms, such as UIQ and S60.

It is important to remember that Pocket PC devices typically have a keyboard and a touch screen. Windows Mobile devices on the other hand are typically non-touch screen with either a keyboard or keypad. These look a lot like UIQ and S60 respectively.

Note that we are not going to consider the impact of those features which are not present on both the Windows Mobile and Symbian OS devices involved in the port (for example, Windows Mobile devices may not all support GPS or the accelerometer APIs), or user experience issues such as connectivity differences.

### 2.1 API History

It is difficult to directly compare the C-based Windows API with the C++ API of Symbian OS, and therefore many of the examples in the paper will look at the MFC (Microsoft Foundation Classes), as these align more naturally with Symbian C++.

Symbian OS was built with C++ from the start, even before Standard C++ was defined. This means that the APIs are C++ classes and exception handling is the norm. Like current Microsoft Windows releases, the APIs are Unicode enabled and work most efficiently with Unicode.

Both MFC and Symbian OS have a rich application layer where there is a lot of pre-wrapped functionality, albeit using different API names and classes. This high level of abstraction makes porting a challenge.

### 2.2 Resources

Symbian OS is a fully pre-emptive multi-threaded operating system and has support for all the standard synchronization objects, such as critical sections and mutexes, that Microsoft Windows does. The main difference is that in the Windows API these are handles, rather than being class-based. In MFC there are classes which provide wrappers over the handles.

Typically, developers write their own data structures when using the Windows C-based API. MFC, however, has a rich collection of type-safe data structure manipulation classes such as arrays, maps and lists. Symbian OS has the same rich data structures, but in different classes. In Symbian

---

<sup>1</sup> Recently renamed Windows Mobile Standard and Windows Mobile Professional.

OS, arrays are handled by either the RArray classes or one of the classes inherited from these base classes. Maps are handled by the RHashMap and RHashSet classes, and lists are handled by the TSgl Queue class. Both platforms now support the Standard Template Library to provide a common library of data structures.

A key problem when moving from the Windows environment to Symbian OS is the handling of strings. The C API uses the Standard C library for handling strings (`string.h`), and MFC uses its own string class called `CString`. Symbian OS, however, has its own hierarchy of string objects and interfaces called descriptors. Descriptors are one of the most fundamental classes in Symbian OS and emphasis needs to be placed on learning them in depth, especially how they are used in place of `CString`. Due to their complex, inheritance-based nature, a full description of these classes and their usage is beyond the scope of this paper; there are additional links to descriptor tutorials in the 'Technical Resources' section.

It is standard to use a client-server model for interaction with system services on Symbian OS. This allows for controlled access to the service. Normally in Symbian OS, a connection is made to a service and the connection is used to create other objects based on that session. Whilst in MFC a `CFile` object would be created, in Symbian OS this requires a connection to the file server (RFs) and then a file object (RFfile) to be opened on the file server session.

## 2.3 Application Structure

Most native code developers on Windows Mobile will be familiar with the Windows message architecture. In Windows, there is a main message loop that waits for a message and, when one is received, dispatches it to the correct window via `GetMessage()/DispatchMessage()` and a `WndProc()` handler for each type of registered window. This allows for asynchronous and synchronous processing using the `PostMessage()` and `SendMessage()` functions.

The equivalent on Symbian OS uses the active object framework, with the active scheduler performing the same role as the message pump.<sup>2</sup> The active scheduler waits for an active object to be signaled and when this happens the active scheduler calls the `RunL()` method of the appropriate active object. It is important to note that the active object framework is, by definition, non-pre-emptive and requires the developer to break long running tasks into a number of states, otherwise the thread may become unresponsive whilst processing a request. This should be very familiar to most Windows developers and the same rule applies during the processing in the `WndProc()` function of the window class instance.

MFC provides for a full application framework, with a model/view/controller type framework for the user interface. This translates easily on to the Symbian OS UIKON UI framework which has classes that are very similar in design to the MFC classes. There is a document object, an Application UI (frame window) object, view and container (CWnd) support. The MFC framework itself is responsible for routing commands in the framework; much like the Application UI does for Symbian OS and its licensees' derivatives.

A list of MFC classes with their corresponding Symbian C++ and S60 APIs is shown in Table 1.

---

<sup>2</sup> Fibers on the Microsoft Windows platform is similar to active objects. However, unlike Symbian OS, the scheduling of the fibers is left to the application developer.

<b>MFC</b>	<b>Symbian</b>	<b>S60 (Licensee extensions)</b>
CWinApp	CEi kAppl i cati on	CAknAppl i cati on
CDocument	CEi kDcoument	CAknDocument
CFrameWnd	CEi kAppUi	CAknAppUi
CDi al og	CEi kDi al og	CAknForm/CAknDi al og
CVi ew	MCoEVi ew	CAknVi ew
CWi ndow	RWi ndow/CCoeControl	
CArray	CArray/RArray/RPoi nterArray	
CLi st	TSgl Que <sup>3</sup>	
CMap	RHashMap <sup>4</sup>	
CCri ti cal Secti on	RCri ti cal Secti on	
CMutex	RMutex	
CSemaphore	RSemaphore	
CDatabase	RDb/RDbNamedDatabase	
CRecordset	RDbVi ew	
CSocket	RSocket	
CI nternetConnecti on	RConnecti on	
CI nternetSessi on	RHttpSessi on/RHttpTransacti on	
CRect	TRect	
CPoi nt	TPoi nt	
CSi ze	TSi ze	
CStri ng	HBufC/TBuf/RBuf <sup>5</sup>	
CTi me	TTi me	
CTi meSpan	TTi meI nterval ....	
CDC	CGraphi csContext <sup>6</sup>	
CFont	CFont	AknFontUti l s
CRgn	RRegi on	
CBi tmap	CWsBi tmap/CFbsBi tmap	CAknI con

**Table 1: MFC - Symbian class comparisons.<sup>7</sup>**

<sup>3</sup> Note that this class always uses dynamically allocated objects.

<sup>4</sup> Not part of the S60 SDK – it was omitted in error.

<sup>5</sup> For more information on the descriptor architecture see the [descriptors FAQ](#).

<sup>6</sup> Symbian does not have handles to individual objects such as brushes; these are methods on the graphics context.

## 2.4 Conventions

One aspect of developing for any platform is to adhere to the User Interface Style Guide specifications of the platform. This promotes ease of use and confidence in the application when it looks and behaves like a native application. Both UIQ and S60 have specific requirements for applications, both in their appearance and usability, and these differ quite substantially from Windows Mobile, especially if the application is touch-enabled and the application is being ported to non-touch variants. It needs to be accepted that the UI experience will differ on Symbian OS due to the different nature of the platforms, the UI controls of the platform and the testing/shipping requirements (for example, 'help' and 'about' commands are required). This will mean new documentation and help files for the port.

Symbian has its own source code conventions which are different to Windows Mobile.

Following the coding conventions for a specific platform can make it easier for new developers to understand the code. It also makes it easier and quicker to identify errors in programming, as coding errors tend to stand out more when the source code looks like the code for the platform. This is particularly important with Symbian OS, where function name suffixes often have meaning. For example, functions suffixed with 'L' can throw exceptions and, unless handled correctly, can leak resources.

Whilst the *Coding Standards* booklet (see 'Technical Resources') covers this, there are three principal types of objects that need to be discussed for Windows developers, particularly for those from an MFC background.

In MFC, almost all objects are derived from `CObject`. These may or may not be allocated on the heap. Simple classes are not derived from `CObject` and have no common base, but these are generally simple structures or lightweight wrappers over simple structures. All classes in MFC are prefixed with a `C` to denote this is a class. MFC implements exception handling correctly, so that when constructors fail, their objects are correctly unwound and no resources are lost.

In Symbian, prefix notation convention is very important as it makes clear the type of resource being dealt with. Classes prefixed with a 'C' must be derived from the Symbian `CBase` class and must be allocated on the heap, never the stack. This forces the class to have a virtual destructor and thus guarantees cleanup for inherited classes when the object is deleted. By allocating the object on the heap this also guarantees the convention that C classes have their member data initialize to zero by the allocator.

Resource owner classes are prefixed by 'R' and are allocated on the stack or as a member variable. These classes are lightweight wrappers over simple structures such as handles and generally have an `Open()` or `Connect()` function to initialize the object and always have a `Close()` function to free the resources. The destructor must never call `Close()`.<sup>8</sup>

All classes prefixed with 'T' are structures and, like R classes, are generally allocated on the stack or as member variables, but they never own resources, and so will never be put on to the cleanup stack<sup>9</sup> and are never required to be deleted.

---

<sup>7</sup> Reference: [msdn.microsoft.com/en-gb/library/ws8s10w4\(VS.80\).aspx](http://msdn.microsoft.com/en-gb/library/ws8s10w4(VS.80).aspx).

<sup>8</sup> This may seem strange, but think of the case where an R class is copied; the compiler will do a lightweight bitwise copy of the object and so this would lead to the object having its resources closed twice.

<sup>9</sup> For more information on the cleanup stack, see the documentation in the Symbian Developer Library [here](#).

## 2.5 Error and Exception Handling

Applications running on Symbian OS devices tend to run for a long time without the need for the device to be restarted or the application terminated, so a high degree of emphasis is put on ensuring that non-fatal errors are handled correctly whilst unexpected and unhandled fatal errors are diagnosed quickly and early on in the development process. This means that, when compared to Windows Mobile, API usage can seem very unforgiving. Fatal programming errors, including issues such as invalid array indices and invalid handle usage, will cause the application to end prematurely with an error. This is called a 'panic' in Symbian OS and the closest equivalent on Windows is a 'General Protection Fault.' The only way to avoid this scenario is to test often and ensure that all errors are handled correctly, including memory leaks and resource leaks. Failure to handle these issues will result in difficulty getting the application through the signing process later on (see Section 5).

As the Windows API is C-based, exception handling is not supported<sup>10</sup> and errors are returned via a return value or via the `GetLastError()` function. In the MFC framework, there is full exception support with the standard C++ try/catch/throw syntax as well as support for the C-based API, depending on whether the MFC wrappers or the native API is used.

Symbian OS has its own error handling architecture called the cleanup stack. The Workbook listed in the 'Technical Resources' section contains a detailed tutorial on using this essential coding mechanism, as do many of the Symbian Press books.

## 3 Getting Started

This section covers development tools, examples and porting tips.

### 3.1 Tools

The SDKs for the two primary Symbian platforms can be downloaded from the Forum Nokia and the UIQ websites. Once the SDKs are installed, an IDE should be installed. The main development IDE is the Eclipse platform-based Carbide.c++, which has replaced CodeWarrior and offers many more features, especially at the professional level. These include on-device debugging, a UI designer, code analyzer tools and performance profiler, all of which add substantially to the quality of the final product. There is also a Carbide.vs Visual Studio plug-in that supports development in both Microsoft Visual Studio 2003 and 2005.

Both free and paid-for versions of Carbide.c++ are available. The Visual Studio plug-in is free. Links to these tools can be found in the 'Technical Resources' section.

### 3.2 Porting Techniques

Porting from Windows Mobile to Symbian OS is challenging as the two operating systems are totally different, both in their style and implementation. Whilst MFC and the UIKON framework have a similar design pattern and style, they are sufficiently different to make UI code reuse almost impossible.

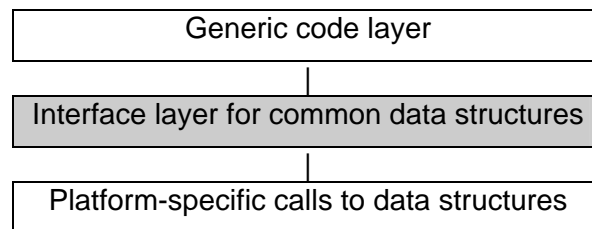
The first part of any port involves identifying areas of commonality and areas of reuse. To successfully make a port, good programming practice recommends that the application be split into a GUI and an engine layer. The engine provides the core application functionality and the GUI provides the user interaction and display.

---

<sup>10</sup> Structured Exception Handling (SHE) is not considered exception handling due to the fact that it is reserved more for kernel type errors and not for general exception handling. This is the same in Symbian C++.

To make code reuse a reality involves pre-planning and possibly the refactoring of the existing Windows code. If there are generic, complex or critical routines that need to be shared across platforms, implement these routines in a DLL and write the functionality using the Standard C libraries and POSIX for system calls. The Open C/C++ plug-ins allow Standard C, Standard C++ and POSIX code to be reused from other platforms, including Linux and Microsoft Windows.

An alternative is to provide an additional layer of abstraction in an interface layer. Common data structures can be defined in a generic way and the actual implementations customized for each platform. The common code layer can then be laid over these libraries.



**Figure 1: Interface layer allowing re-use of generic code on multiple platforms.**

### 3.3 Porting Console Applications to Symbian OS

Unlike Windows, where creating a console application is a more a matter of setting a linker flag, Symbian OS requires that the asynchronous and exception handling frameworks are created before an application can be started.

Thus the typical console application looks like the following (and this is before any code has been written):

```

void MainL()
{
    CConsoleBase* console = Console::NewL(KTextConsoleTitle,
                                         TSize(KConsFullScreen, KConsFullScreen));

    CleanupStack::PushL(console);

    // Create an active scheduler. A very simple console application might
    // not need one, and could simply execute its functionality here
    CActiveScheduler* scheduler = new(ELeave) CActiveScheduler;
    CActiveScheduler::Install(scheduler);

    // call the main function under a TRAP harness. The main function needs to:
    // add an active object, mark it ready to run and start the active scheduler
    TRAPD(mainError, DoMainL(console));

    delete scheduler;

    if (mainError)
        console->Printf(KTextFailed, mainError);
    console->Printf(KTextPressAnyKey);
    console->Getch();

    CleanupStack::PopAndDestroy(console);
}
  
```

```

GLDEF_C TInt E32Main()
{
    __UHEAP_MARK;
    TInt err = KErrNone;

    CTrapCleanup* cleanup = CTrapCleanup::New();
    if(cleanup)
    {
        TRAP(err, MainL());
        delete cleanup;
    }
    __UHEAP_MARKEND;
    return err;
}

```

As can be seen, this is a substantial deviation from a console application written in C/C++ on Windows. Windows applications use the standard `main()` function and the linker provides the console behind the scenes together with the handling of key presses and displaying the text.<sup>11</sup>

Generally speaking, when migrating console applications such as test harnesses from Windows to Symbian OS, it may be preferable to use the Open C/P.I.P.S. libraries. This will make the porting step much easier, especially if the code is already POSIX/C/C++ compliant. For example, `std::string()` will work unchanged when ported, but a console application using `CString` (found in MFC) will not work directly as it contains code specific to MFC.

### 3.4 Porting a GUI Application to Symbian

Unlike console applications, porting GUI applications will be more of an effort as both the Windows APIs and the frameworks above it (MFC and ATL/WTL) both have substantially different APIs to Symbian OS due to their high level of abstraction.

It is not practical to include the entire source of a simple 'HelloWorld' here, as it covers multiple source and resource files. To generate a simple application, from the 'File' menu in Carbide.c++ v1.3, select 'New' and then in the submenu select 'Project.' This will display a dialog that allows various projects to be created, and from this select 'Symbian OS' and 'Symbian OS C++ Project.' This displays the various wizards available and, if the '3<sup>rd</sup>-Future Ed. GUI Application' is selected, it will generate the skeleton of a 'HelloWorld' application.

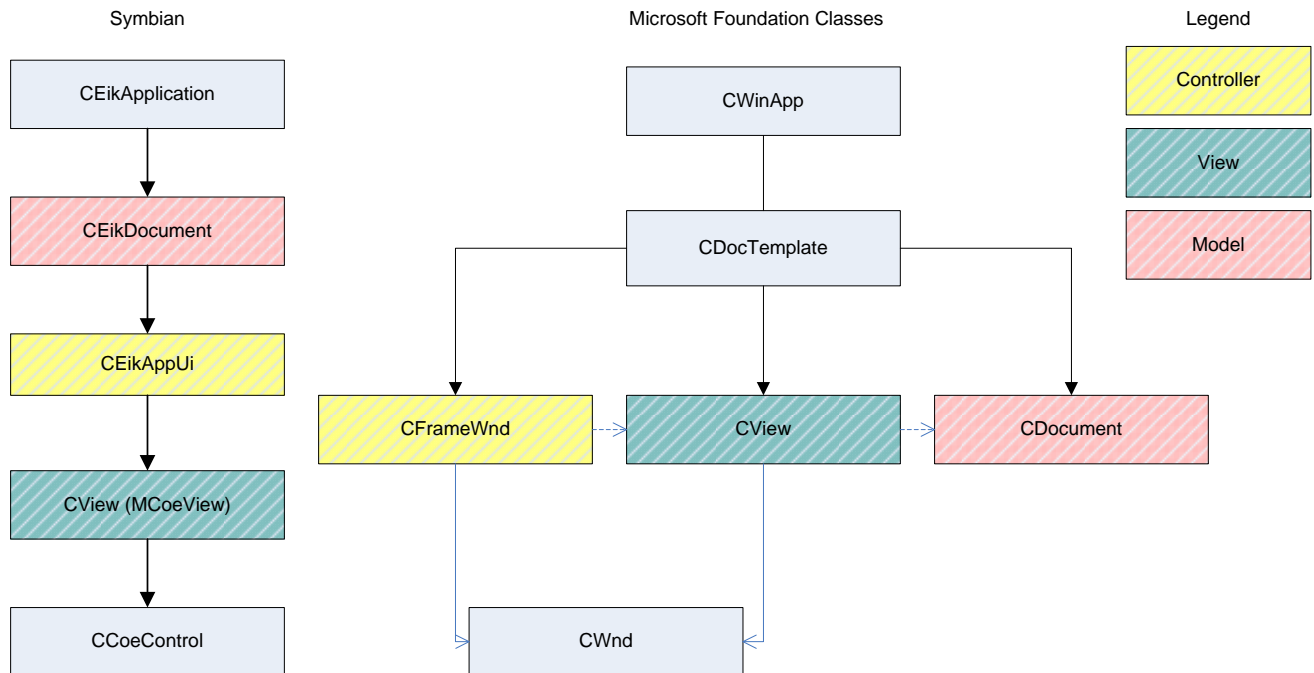
Whilst a Symbian OS graphical application without a framework is possible, it is very difficult to implement all the required pieces of the application framework in practice. Instead, it is preferable that applications be built on the supplied framework. Developers should identify which portions of the application relate to the GUI and which to the engine, and split the application accordingly. It is extremely difficult to 'retro fit' a GUI into a console type application.

Symbian OS contains an application framework, UIKON, which provides a set of base classes. When Symbian licenses their operating system to third-parties, the base classes of the application framework are extended and customized by licensees to their own requirements; for example, Nokia customize UIKON to the AVKON framework and UIQ customize this to the QIKON framework. The developer will therefore find two layers of inheritance on top of the base classes: one for the licensee customization (e.g., AVKON) and one for their own implementations of the classes, itself derived from the licensees' classes.

---

<sup>11</sup> There are some good references on the Forum Nokia website wiki about using the console in the emulator and on a real device, as there are significant issues that need to be addressed to have it work both on a device and emulator.

Due to its C++-based architecture, a typical Symbian OS application will be composed of a number of C++ classes using the 'Model View Controller' (MVC) design pattern (as shown in Figure 2).



**Figure 2: Class relationships.**

This is the same model that is followed in MFC: there is an application object (CWinApp), a document object (CDocument/CDocTemplate), an Application UI (CFrameWnd), views (CView) and they all operate in the same way the MVC pattern does. Commands and messages are routed by the application via the command handlers in the frame, view and document objects. The ultimate result is that the container control (CCoeControl) inside a view is the recipient of the display requests and handles input from the user. The CCoeControl performs the same function as the CWnd in MFC. Because Symbian OS does not support a MDI (multiple document) framework, each window is tied to a view and the view is tied to the Application UI. There is only one view active at one time and within that view will be a container. The MFC Document Template class links the document, frame window and view together, whilst in Symbian OS this is done via the view being added to the Application UI.

## 4 Porting Tips

This section covers the most common development issues that developers moving from Windows Mobile to Symbian OS encounter.

### 4.1 General

- Some devices are touch screen enabled and some are not (on both platforms). If the port is from a touch screen device to a non-touch device, then consideration needs to be given as to how this should be handled. Likewise, the migration could be from a QWERTY keyboard to a keypad and again UI issues need to be considered.

- Symbian OS, unlike other platforms, discourages the use of static writable data in DLLs and so applications that rely on it will need to reconsider how they plan to use global variables in their applications. There are a number of solutions to this problem available and some are covered in the white paper on the Symbian Developer Network website.<sup>12</sup>
- Installing an SDK for Symbian OS requires a number of additional toolkits to work, such as the version of Perl needed by the Symbian OS toolchain. There are also two compilers available for Symbian OS: the free GCCE version, which is the open source compiler and linker for the Symbian OS toolchain, and the commercial RealView toolkit (RVCT) from ARM.
- Active objects should be used in preference to multiple threads and synchronization objects. This is because a context switch to another thread incurs relatively large power consumption overheads. Often, Windows developers come from a multithreaded background and think that active objects can be avoided by using threads. Due to the way the Symbian OS kernel works, talking to other system services will almost certainly require a TRequestStatus signal object to be used at some point. Whilst a thread can be spawned off to deal with the request, handles are normally all thread-relative and cannot be easily shared between processes, so an active object is almost certainly required (which necessitates an active scheduler too).

It is also very important to note that the GUI runs in the main thread of the process and access to any UI functionality must be done in this thread; for example, updating a control or displaying a dialog box cannot be done from another thread. These observations suggest that active objects should almost certainly be used, and that the user interface runs in and is only accessible in the primary thread of the process - other threads will require some mechanism to report their changes back to the main GUI thread to update it.

- In Windows Mobile, each window is a real window with an associated handle and class instance. In Symbian OS this does not hold true; each view has a single container and that container generally has one real window and multiple container windows. This means that container windows are not available to other processes to enumerate and discover. Likewise, two common techniques that developers use in Windows are to attach pointers to the spare 32 value in each window instance (via `GetClassLong()/SetClassLong()`) and to use the text value associated with each window handle (via `GetWindowText()/SetWindowText()`). Neither of these is supported due to the container window-based nature of the Symbian OS window server.

## 4.2 Good Design Practices

It is good practice to split an application into a user interface and an engine. The UI performs the interaction with the user and the engine does the processing and storage. Historically, in Windows this was done by putting the logic into a DLL and then exposing an API through the DLL. The downside to this approach is that when state needs to be shared between multiple processes, it is difficult to do easily. Symbian OS encourages the GUI/engine model and goes further with its rich IPC layer where the logic can be encapsulated into a thread-safe separate process if need be.

---

<sup>12</sup> [developer.symbian.com/main/downloads/papers/static\\_data/SupportForWriteableStaticDataInDLLs.pdf](http://developer.symbian.com/main/downloads/papers/static_data/SupportForWriteableStaticDataInDLLs.pdf).

### 4.3 Component-based Architectures

One of the most used architectures within Windows Mobile is the Component Object Model (COM) objects. The COM runtime provides methods for applications to discover and use objects installed by third-parties or enhancements added after the product has shipped.

Each COM object has an interface and a COM class associated with it; the class holds the implementation and the list of interfaces the object supports. Each interface is identified by a GUID. Each object implements one or more interfaces, but the object itself is never visible, only its interface.

ECOM is the Symbian OS implementation of the component object model. ECOM provides a rich framework to identify and search for objects that implement a particular interface. It is normal to find a number of objects installed, all registered for the same interface UID, but in different DLLs. The caller then selects the appropriate instance UID from the list of instances for the particular interface.

ECOM differs significantly from COM. Every interface in COM is derived from `QueryInterface` and all reference counting is maintained by the object. ECOM exposes a base class with virtual methods to the caller and the actual implementation implements these methods. Multiple interfaces within an object are not supported. When an object is created by the ECOM runtime, the implementation of the object is constructed and the base object returned. The object manages its own lifetime and the ECOM runtime determines the lifetime of the loaded DLL implementing the object.

### 4.4 Security Models

Probably the most difficult aspect of porting to Symbian OS is learning how security is implemented, as Symbian OS has a much stricter platform security model than Windows Mobile.

Windows Mobile only requires that the application be signed with a certificate; there is no concept of data caging and a very limited model for restricting application access to APIs. There are two levels of APIs, one for normal applications and one for privileged applications. Normal applications are barred from accessing certain APIs such as telephony APIs and SIM APIs.

Within Symbian OS, APIs are grouped according to their functionality and how critical they are to the overall functioning of the system. Access to each group of APIs is controlled by an access permission known as a 'capability'; capabilities are defined by the developer at compile time and enforced during installation and runtime.

In addition to capability protection, Symbian OS has additional features to ensure that applications run in a safe and secured manner:

1. All executable code must go in the `\sys\bin` folder.
2. Each executable and DLL file must have a unique system-wide name. Only a package with the same identifier may replace or update existing DLLs or executables. This is particularly important in the case where third-party libraries are to be used, and means that any third-party library needs to be put into its own SIS file and signed so that it can be shared across applications and independently upgraded.
3. Each executable or DLL file has associated with it a set of capabilities defined by the developer at compile time and enforced during installation and runtime. There are a simple set of rules to ensure that security is never compromised (i.e., to prevent the DLL from 'gaining' capabilities):
  - a. An executable can never load a DLL with fewer capabilities than itself.

- b. A dynamically loaded DLL must have at least the same capabilities as the application loading it.
4. Each executable has associated with it a UID that uniquely identifies it and is obtained from the Symbian Signed website.<sup>13</sup> The application has a private data area in `c:\private\<application UID>` where the application may store private or sensitive data; the folder is not accessible to other applications. This is termed 'data caging.' If the application needs to share data between two or more processes, it is recommended that the shared data be moved to a separate server application that both applications can access or the data can be put into a public folder so that both applications may read it.

If the released application is to run on all phones without issuing warnings, it needs to go through the Symbian Signed process. This is roughly equivalent to the processes required by the Microsoft Mobile 2 Mobile program, but unlike M2M, it is mandated that the application be Symbian Signed. Further details on releasing applications are in Section 5; more information about platform security is listed in the 'Technical Resources' section.

## **4.5 Localization, Internalization and Input**

Windows Mobile and Symbian OS both support multiple languages, locales and input methods. Both handle internationalization of their interfaces via the native operating system. This means that changes (such as bi-directional text) are largely hidden from the user unless they specifically provide their own controls, in which case it is up to the licensee to provide APIs to handle the globalization.

The locale (country) handling is similar on both platforms and depends on the language the device is currently in as to how numbers, dates and times are formatted.

On Windows Mobile, the main method to input non-Latin characters is the IME or input method editor. This allows developers to extend the inbuilt input methods to provide a custom means of entry such as via alternative key mappings on a virtual keyboard or via a handwriting recognition module. Symbian OS has its own input method API called a Front End Processor (FEP) that allows the handling of characters to be done by another application before they are delivered to the active application. A FEP also allows for the developer to provide handwriting recognition and predictive text input, taking the input and passing new character(s) to the current application.

Windows Mobile stores resources as part of the executable and not as a separate file. Normally, developers select and load a DLL to get the appropriate resources for the current language. Symbian OS has a resource file for each language and selects the appropriate resource file for the current language automatically (via the UI framework). If extra resource files are being used by the application, these should be loaded using the `BaflUtils::NearestLanguage()` function to ensure the best language is chosen from the list of available language resource files. The toolchain will handle compiling the resources for each language and the device will handle the selection of the appropriate language for the user.

## **4.6 The Registry**

The Windows registry holds all the system settings, including those both for the device and for the applications. Symbian OS does not have a concept of a third-party general settings API in the public SDK. It is recommended that if settings are to be used, the developer should create a new Symbian server and use this to control access as to who can read and write the settings. The

---

<sup>13</sup> See [www.symbiansigned.com](http://www.symbiansigned.com).

application can then talk to the settings server to get the data. This is obviously a rather heavyweight process and involves lots of code; however, for settings that do not need to be protected or those that do not need to be shared, it is simpler just to put a settings file in the data caged directory and then read the settings from the file when needed. If multiple applications need to access the data, the file could be put into a shared public area and the settings all read from there.

## 4.7 Services

Windows has a limitation of 32 processes per device. This means that care must be taken not to exceed these limits, otherwise new applications will be unable to start. This is especially important in the case where the executable is actually a service component that is providing functionality to other applications. In later releases of Windows Mobile, this issue was addressed through the services API, which allows third-parties to build a service as a DLL and have this accessible through a standard, albeit simple, API. This meant that the services executable could now hold all the system services, freeing up slots in the process table to allow more applications to run simultaneously.

Symbian OS has a very rich IPC API based around the client-server model. The Symbian OS kernel provides services as executables, which allows for 255 separate processes. Access to the functionality of the server is governed by the application's UID and capabilities, allowing a client application to talk safely and securely to a server. Each server is generally run as a separate executable<sup>14</sup> without a user interface as this allows control over which applications can use which functionality. For example, the telephony server can block requests to dial a phone number if the calling application is not capable of using network services.

## 4.8 Binary Compatibility

Each Windows Mobile executable and DLL library exports a list of functions it exposes to the world and a list of functions it needs from other DLLs. These are termed the export table and import table respectively, and are stored as an array of function entry points. To access the function, you either need to know the index or ordinal within the array where the function's entry point is stored, or you may be given the function's name and must use another table in the DLL to look up the ordinal. Functions can therefore be linked by name; the developer does not care if the ordinal changes between releases or not so long as the function name and argument list do not change. The loader will fix these up when the process is created.

Symbian OS, on the other hand, strips out the function name mapping from the DLL and only allows the functions to be linked by ordinal. This simplifies the loader code and also reduces the size of the final executables and DLLs. This means that in Symbian OS, applications need to know the ordinals at link time; also, ordinal values cannot be changed after a project is deployed, otherwise any applications depending on the DLL will have an incorrect import table. The method Symbian OS uses to control binary compatibility is outside the scope of this paper, but the 'Technical Resources' section has a link to additional documentation that discusses how it should be used.

In Windows Mobile, there are very few issues with linking, even between major releases, because the DLLs are exported by name. In contrast, Symbian and its licensees have to take care to ensure that all APIs published in the SDK will work for future SDKs, even within the same major release of

---

<sup>14</sup> It is possible to run multiple threads and have a server on each thread; this technique allows for multiple servers to share one private data folder to share settings, for example.

the operating system. To ensure the application will run on the widest set of supported devices, a decision needs to be taken early on to identify the lowest supported device in the set of supported devices. In context, 'lowest' means the lowest minor version number. For example, a device such as the N73 runs S60 3rd Edition (S60 3.0) and the N95 runs S60 3rd Edition, Feature Pack 1 (S60 3.1). If the application is built using the S60 3.0 SDK, the application will work unchanged on the N95. If, however, the application is built with the 3.1 SDK it may or may not work on a 3.0 device. For example, if the Profile API is used (which was only introduced in 3.1) then the application will not run on a 3.0 device.<sup>15</sup>

## 5 Releasing the Application

Unlike Windows Mobile, where the end user can, for the most part, disable all security, Symbian OS places more emphasis on security, as discussed previously. Additional steps therefore need to be followed to test and release an application and those steps depend on the capabilities required by an application.

### 5.1 Testing

When developing applications that use capability-protected APIs, testing on actual devices before they are Symbian Signed may require a developer certificate. You can sign an application for testing on a single device (without requiring a Publisher ID) by visiting the Symbian Signed website and using the 'Open Signed' option.

Generating a developer certificate for more than one device requires a Publisher ID to be obtained from TC TrustCenter. Each developer certificate is limited to the IMEI numbers specified when the developer certificate was requested. This differs from Windows Mobile, where a certificate is generated on the fly and the device can be unlocked by the operator very easily.

### 5.2 Signing

Applications using non-capability-protected APIs (around 60% of Symbian OS APIs) will need to have the application self signed with a key generated by the user; Carbide.c++ can do this automatically. In this situation, the user will be informed during installation that the application is not trusted, and the features that it may access are listed. (Note that this policy may be changed by device manufacturers or network operators in future.)

Applications requiring access to APIs protected by more restricted capabilities require the application to be externally verified and signed via Symbian Signed. If APIs protected by manufacturer-approved capabilities are being used, the developer will need to contact a licensee to obtain the necessary authorization before the application can be signed.

Symbian Signed offers a number of signing options, varying from signing for single devices, freeware and open source signing, to Certified Signing for commercial distribution. Further details about all the options available through Symbian Signed are discussed in the *A Guide to Symbian Signed* booklet listed in the 'Technical Resources' section.

There is also more general information on releasing Symbian OS applications in the *Getting to Market* booklet listed in the 'Technical Resources' section.

---

<sup>15</sup> Newer devices based on Symbian OS support a mechanism called Feature Manager, which enable application developers to query the device in runtime for existence of a specific feature and use it, if present.

## 6 Useful Tools

Probably the most useful tool to assist in debugging is HookLogger. This application allows the trapping of memory and resource leaks as well as isolating deadlocks in the emulator. This tool has recently been updated for Symbian OS v9.x to include a new tutorial and examples.

A developer certificate containing the list of valid IMEIs is required to sign a SIS file. If a Publisher ID is available, the developer certificate request tool can be used to generate a certificate request for more than one device. This is freely downloadable, and there is a version available for integration into the Carbide.c++ IDE.

One of the requirements of Symbian Signed is to minimize power consumption. For S60 3rd Edition Feature Pack 1 devices (and onwards), Nokia provide an Energy Profiler to show the battery consumption on the device and for an application.

See the 'Technical Resources' section for links to these tools.

## 7 Summary

In this paper I have discussed the differences between application development for Microsoft Mobile OS and Symbian OS. It is much easier to understand the many differences between the two when the frameworks are compared, rather than the actual APIs. The Microsoft Foundation Classes and the Symbian application framework both owe a lot to the Model View Controller design pattern and once the design of the first one is understood, it is easier to comprehend the design decisions taken for second.

With any project, good design will make or break it and quite often a full rewrite is not needed. By doing simple things like Test Driven Development (TDD), design patterns, splitting the User Interface and logic into separate components and using third-parties' libraries where appropriate, time to market can be significantly reduced. As Symbian moves more towards a Standard C/C++ framework it is becoming increasingly easier to have common sets of engine code shared across the two platforms with minimal changes on both. This should borne in mind when refactoring or designing new code on either platform.

Symbian has come a long way in the 10 years that it has been around and there are now world class development environments, technical support and tools to help development efforts, as well as forums with dedicated and helpful people throughout the world to assist you, so it's definitely not as scary as it is often thought to be.

## 8 Glossary

<b>Term</b>	<b>Definition</b>
AO	<i>Active Object</i>
AS	<i>Active Scheduler</i>
ATL	<i>Active Template Library; this is has been merged into MFC.</i>
COM	<i>Component Object Model</i>
DevCert	<i>Developer Certificate</i>
ECOM	<i>EPOC Component Object Model</i>
GUID	<i>Globally Unique Identifier</i>
IMEI	<i>International Mobile Equipment Identity</i>
IPC	<i>Inter-Process Communication</i>
M2M	<i>Mobile To Mobile</i>
MFC	<i>Microsoft Foundation Classes</i>
MVC	<i>Model View Controller design pattern</i>
OS	<i>Operating System</i>
SVG	<i>Scalar Vector Graphic</i>
UID	<i>Unique Identifier</i>
UIKON	<i>Symbian's OS framework, formally called EIKON</i>
WTL	<i>Windows Template Library</i>

## 9 Technical Resources

The links provided here are valid as at time of writing.

### **Introductory references**

[Symbian OS Basics Workbook](#)  
[Descriptors FAQ](#)  
[Descriptors presentation](#)  
[Descriptors Example](#)

### **Security and Signing references**

[A Guide to Symbian Signed](#) (booklet)  
[Platform Security for all](#) (booklet)  
[Symbian Signed Test Criteria](#)  
[Symbian Signed website](#)

### **ECOM references**

[ECOM Example](#)  
[ECOM Plugin Architecture](#)

### **IPC references**

[Transient Server Template](#)  
[Transient Server source code](#)  
[New IPC Mechanisms for Symbian OS](#)

### **Binary Compatibility references**

[How to control Binary Compatibility](#)

## Tools

[S60 SDK Download](#)

[UIQ SDK download](#)

[HookLogger](#)

[Carbide](#)

[Developer Certificate Request](#)

[Symbian Recommended Tools](#)

[Energy Profiler](#)

[Open C/C++](#)

[P.I.P.S. wiki](#)

## 9.1 Websites

[Symbian Developer Network](#)

[Forum Nokia](#)

[NewLC](#)

[Sony-Ericsson](#)

[UIQ](#)

[S60.com](#)

## 9.2 Books

[Symbian Press booklets](#), including *Getting to Market* and *Coding Standards*

[Symbian OS Explained](#)

[Symbian OS Internals](#)

[Symbian OS Platform Security](#)

## 10 Author Profile



Paul first started developing games on a Commodore 64 whilst at school a long time ago in South Africa.

After finishing university, his first experience with Symbian was one of the first Psion devices, a Psion LZ64 which he used to interface to surveying instruments. The data collected there was exported into Autocad and used to generate maps for civil engineers.

In the mid 1990s Paul immigrated to England, where he worked on quantity surveying tools until becoming involved with one of the first enterprise sync tools for Mobile devices (ASL Connect).

After six years of working on Windows Mobile/CE, Paul moved over to the Symbian team on the departure of one of the Senior Symbian developers.

Now he designs enterprise class software for PIM Data Management (Onebridge) and device management (Afaría), and is looking at ways development can be made both faster and more reliable so that the end products provide a better end user experience out of the box.