

# Writing Good Symbian OS Applications

Version 2.04  
ISSUED  
14<sup>th</sup> June 2005

## Contents

CONTENTS .....	1
HISTORY .....	2
INTRODUCTION .....	2
GENERAL TIPS .....	2
DESIGN TIPS .....	2
CODING TIPS.....	2
TESTING TIPS.....	5
DEBUGGING TIPS .....	5

## History

Version	Date	Details
v1.00	27 <sup>th</sup> November 2003	First issue to Symbian Developer Network
v2.00	16 <sup>th</sup> February 2004	New release incorporating feedback from other Symbian staff
v2.01	29 <sup>th</sup> April 2004	Added tip about not hard-coding file locations
v2.02	10 <sup>th</sup> August 2004	Minor tweaks
v2.03	10 <sup>th</sup> November 2004	Minor tweaks after feedback from Symbian staff
v2.04	14 <sup>th</sup> June 2005	Added recommended practice for SIS file version information

## Introduction

When writing applications for Symbian OS there are many factors to consider – from the design right through to the final finishing touches of releasing your .SIS file – which all contribute to the quality and robustness of your application. This document aims to gather together some useful hints, tips and links which you, the developer, can use to produce the most reliable Symbian OS applications possible.

## General Tips

1. The Symbian Developer Network hosts a lot of valuable information to help you write applications for Symbian OS. Visit <http://www.symbian.com/developer> regularly to obtain the latest SDKs, technical information, code examples and white papers. You should also sign up to the [Symbian Developer Network newsletter](#) – this is distributed monthly by e-mail and is the best way to keep up to date with the latest Symbian developer news.
2. Symbian's licensees also run developer-related programs. You should register at their sites to get access to the latest phone-specific information and tips.
3. Hosted on the Symbian Developer Network website, the [Symbian OS FAQ database](#) is an invaluable source of information for developers covering the most frequently asked design and coding issues. The [Symbian OS C++ Coding Standards](#) are also hosted there. In the course of developing Symbian OS itself, Symbian has defined several important coding idioms and styles. The [Coding idioms paper](#) explains them for 3<sup>rd</sup> party developers. By using these tried and tested conventions, you will benefit from all of the experience Symbian has gained in working with Symbian OS.
4. Finally, there are a growing number of Symbian OS publications available. [Symbian Press](#) offer several books on a range of topics, all designed to help you write Symbian OS code more easily and reliably.

## Design Tips

1. The most important design tip for Symbian OS is to separate out your “engine” and UI code into different modules. Symbian OS itself is designed this way and it aids porting between different UI systems.

One approach is to make this distinction at the binary level by placing all non-UI related code into a separate engine .DLL file. Your Application UI can then link to this .DLL for access to engine functionality. For example, code that opens up a connection to the Agenda Server (RAgendaServ) uses no UI components and is therefore clearly engine code. The Agenda Server is common to all Symbian OS releases since v6.0 so your engine code would work on all phones without modification. An alternative approach is to make this distinction at source level so that your application builds as one single .APP file but your ‘engine’ and UI-related code is separated into different .CPP and .H files for easier management and debugging.

By coding this way, you ease the porting burden for new UI platforms. As illustrated above, pure engine code often runs unaltered on any UI platform. This means all you have to port and optimize for a new UI is your own, separate UI layer.

2. Always design with support for localization in mind. Never hard-code strings or literals into your source files – always use the resource file mechanism Symbian OS provides to store strings.
3. Take care to stick to using APIs which are documented and supported for given SDK and Symbian OS releases. The use of unsupported or deprecated APIs can lead to future problems for your application – for example, Symbian reserves the right to change or remove APIs which are not intended to be used by external developers.
4. Do not assume all ‘system’ files will be present on all phones. Symbian OS licensees have a great deal of flexibility to customize their handsets so phones based on the same platform but from different manufacturers may contain different support files – so, for example, hard-coding the location of a sound clip you wish to play will not guarantee good portability. At the very least, you should always consider proper handling of the error condition should the file *not* be present on future handsets.

## Coding Tips

The following are a collection of general tips which you should bear in mind when actually writing your code.

1. Ensure your application responds to system shutdown events. It is vital that you respond to `EEikCmdExit` (and any platform-specific events, for example `EAKnSoftkeyBack` on Series 60) in your `AppUi::HandleCommandL()` method.

2. Respond to incoming system events. Remember that your application is operating on a multitasking phone system. You need to pay careful attention to focus gained/lost events, etc. to ensure you respond correctly when the user receives a high priority notification. For example, ensure you save your state and data when there is an incoming phone call which will cause your application to lose focus (i.e. you need to appropriately act on standard 'to background' events – see the SDK for more details). Generally, the framework should handle this for you and no special action is required on your part – but be sure to check you're not doing anything which prevents the framework operating as intended.
3. Memory handling on Symbian OS is a major topic for consideration. Note that behavior on the phone can sometimes differ to that on the emulator. It is **vital** that you test your application on the real phone before submitting it for testing.
4. KERN-EXEC 3 crashes are often symptomatic of stack corruption/overflow - as per the Symbian SDK recommendations, prefer the heap to the stack.
5. Under low memory situations, failing gracefully is important - an application panic indicates a real bug in your code. Here are some key, common errors:
  - Failure to have properly added a non-member, heap-allocated variable to the `CleanupStack`.
  - The 'double delete' - e.g. failure to correctly `Pop()` an already destroyed item from the `CleanupStack`, causing the stack to try and delete it again at a later time.
  - Accessing functions in variables which may not exist in your destructor, e.g.

```
CMyClass::~CMyClass()
{
    iSomeServer->Close();
    delete iSomeServer;
}
```

should always be coded as:

```
CMyClass::~CMyClass()
{
    if (iSomeServer)
    {
        iSomeServer->Close();
        delete iSomeServer;
    }
}
```

- Putting member variables on the `CleanupStack` - never do this; just delete them in your destructor as normal.

6. Always use `CleanupClosePushL()` with 'R' classes which have a `Close()` method. This will ensure they are properly cleaned up if a Leave occurs. For example:

```
RFile file;
User::LeaveIfError(file.Open(...));
CleanupClosePushL(file);
...
CleanupStack::PopAndDestroy(&file);
```

7. In addition, remember that the `CleanupStack` is an extensible mechanism that can be used for cleaning up *anything* when there is a leave. If you have a more complex situation to deal with, don't just ignore proper clean up. See the SDK documentation on `TCleanupItem` for more information. This can be used to ensure proper cleanup of other 'R' classes which do not have `Close()` methods (for example, ones which have `Release()` or `Destroy()` instead).
8. Always set member `HBufC` variables to `NULL` after deleting them if you intend to re-allocate to the same variable. Since `HBufC` allocation (or re-allocation) can potentially Leave, you could find yourself in a situation where your destructor attempts to delete a `HBufC` which no longer exists. This is true of any heap-allocated variable, but doing this with `HBufC` tends to be a common usage pattern.
9. If you have cause to use a `TRAP()` of your own, do not ignore all errors. A common coding mistake is:

```
TRAPD(err, DoSomethingL());
if (err == KErrNone || err == KErrNotFound)
{
    // Do something else
}
```

This means all other error codes are ignored. If you must have a pattern like the above, leave for other errors:

```
TRAPD(err, DoSomethingL());
if (err == KErrNone || err == KErrNotFound)
{
    // Do something else
}
else
{
```

```
User::Leave(err);
}
```

10. Do not wait to `PushL()` things on to the `CleanupStack`. Any newly allocated object (except member variables) should be added to the stack immediately. For example, the following is wrong:

```
void doExampleL()
{
    CSomeObject* myObject1=new (ELeave) CSomeObject;
    CSomeObject* myObject2=new (ELeave) CSomeObject;
    ...
    // Do something here with the variables
    ...
    CleanupStack::PopAndDestroy(2); // myObject2, myObject1
}
```

because the allocation of `myObject2` could fail, leaving `myObject1` 'dangling' with no method of clean up. The above should be:

```
void doExampleL()
{
    CSomeObject* myObject1=new (ELeave) CSomeObject;
    CleanupStack::PushL(myObject1);
    CSomeObject* myObject2=new (ELeave) CSomeObject;
    CleanupStack::PushL(myObject2);
    ...
    // Do something here with the variables
    ...
    CleanupStack::PopAndDestroy(2); // myObject2, myObject1
}
```

11. Note that functions with a trailing 'C' on their name (for example, a `NewLC()` method) *automatically* put the object on the `CleanupStack`. You should **not** push these objects onto the stack yourself or they will be present twice. The trailing 'C' functions are useful when you are allocating non-member variables.
12. Two-phase construction is a key part of Symbian OS memory management. The basic rule is that a Symbian OS constructor or destructor must never Leave. If a C++ constructor leaves, there is no way to cleanup the partially constructed object because there is no pointer to it. For this reason, Symbian OS constructors simply instantiate the object, which then provides a `ConstructL()` method where member data can be instantiated. If `ConstructL()` Leaves, the standard destructor will be called to destroy any objects which have been successfully allocated to that point. It is essential that you mirror this design pattern to avoid memory leaks in your code. For each line of code you write, a good question to ask yourself is, "can this line leave?" If the answer is "Yes", then think, "Will all resources be freed?"
13. Do not use the `_L()` macro in your code. This functionality has been deprecated since Symbian OS v5 – you should prefer `_LIT()` instead. The problem with `_L()` is that it calls the `TPtrC(const TText*)` constructor, which has to call a `strlen()` function to work out the length of the string. Whilst this doesn't cost extra RAM, it does cost CPU cycles at runtime. By contrast, the `_LIT()` macro directly constructs a structure which is fully initialized at compile time, so it saves the CPU overhead of constructing the `TPtrC`.
14. When using descriptors as function parameters, use the base class by default. In most cases, pass descriptors around as a `const TDesC&`. For modifiable descriptors use `TDes&`.
15. Active Objects are a key piece of Symbian OS functionality. You should carefully study the SDK documentation and Symbian Developer Network white papers to get a good understanding of the way these work. Here are a few useful tips:
- There is no need to call `TRAP()` inside `RunL()`. The Active Scheduler itself already TRAPs `RunL()` and calls `CActive::RunError()` after a leave
  - To this end, you should implement your own `RunError()` function to handle leaves from `RunL()`
  - Keep `RunL()` operations short and quick. A long-running operation will block other Active Objects from running
  - **Always** implement a `DoCancel()` function and **always** call `Cancel()` in the Active Object's destructor.
16. Similarly, you should make use of the Active Object framework wherever possible. Tight polling in a loop is highly inappropriate on a battery-powered device and can lead to significant power drain. Pay particular attention to this when writing games – for example see the technical paper on the Symbian Developer Network website [here](#) with a more detailed discussion [here](#).
17. ViewSrv 11 panics are a hazard when writing busy applications, for instance games. They occur when the ViewSrv active object in your, or any other, application does not respond to the view server in time. Typically 10-20 seconds is the maximum allowed response time. See FAQ-0900 for a more detailed explanation and FAQ-0920 for practical tips to avoid this problem. Both are available from the Symbian OS FAQ database at <http://www3.symbian.com/faq.nsf>
18. You don't need to use `HBufC::Des()` to get into a `HBufC`. All you have to do is dereference the `HBufC` with the `*` operator – this is particularly relevant, for example, when passing a `HBufC` as an argument to a method which takes a `TDesC&` parameter (as recommended above).
19. When making use of the standard application .INI file functionality (i.e. by using the `Application()->OpenIniFileLC();` API in your Application UI class), be sure to write into streams with version number information. This allows you to create new streams for future

versions of your application and means if an end user installs a new version of your product in the future, an 'old' .INI file will not cause the new version to panic if it cannot find the right settings or stream.

20. Take care when implementing framework classes in your application. You should always derive from the platform-specific framework classes provided. For example, on UIQ, do not derive your AppUi class from `CEikAppUi`, derive from `CQikAppUi`. All of the application base classes (`CQikAppUi`, `CQikApplication`, and `CQikDocument`) add functionality that the wider framework relies on to make applications perform correctly.

## Testing Tips

1. The most important testing tip is to exit your application under the emulator, **not** just to close the entire emulator. In debug mode, there is memory and handle checking code surrounding the application framework shutdown functions. If you exit your application, this code will be invoked and you will be able to see if you have leaked memory or left a handle (e.g. an 'R' object) open. For UIQ applications it is customary to provide an Exit menu item **in debug mode only** for this purpose.
2. Another vital tip is to ensure the correct platform dependency information is included in your .PKG file prior to deployment. More details of the dependency string you require should be in your platform-specific SDK. FAQ-0853 on the Symbian OS FAQ database at <http://www3.symbian.com/faq.nsf> also offers useful information.
3. When writing your .PKG file, also ensure you use the !\ syntax where appropriate. In general, your application should install and run from any drive on the end user's phone. Very few things (e.g. .INI files) should be put on C:\ only.
4. Ensure your PKG file specifies the version information exactly as you intend it to be. Specifically, double-check the minor version number section – as a guideline, always specify two digits for your minor version to avoid confusion. For example:
  - a. 1,10,7 = v1.10 (build/revision number 7)
  - b. 1,1,7 = v1.01 (build/revision number 7) - i.e. this is **not** the equivalent to (a) above
  - c. 1,01,7 = v1.01 (build/revision number 7) - i.e. this is equivalent to (b) above

## Debugging Tips

1. When writing and debugging a new control class, put `iEikonEnv->WsSession().SetAutoFlush(ETTrue);` in the `ConstructL()` function of your AppUi. This means that `gc draw` commands will show up in the emulator immediately, rather than when the window server client buffer is next flushed. This means you can step through the draw code and see the effect each line is having. However, you must ensure this line does not make it into released software as it has efficiency implications.
2. You should regularly run the LeaveScan tool over your source files. This will check all functions for code which can Leave and report an error if they do not have a trailing L on their name, highlighting a potential bug or oversight in your source. This is useful for checking which code should be allowed to potentially Leave and making sure you handle the eventuality properly. See FAQ-0291 on the Symbian OS FAQ database at <http://www3.symbian.com/faq.nsf> to download the tool and read further information.
3. If your application panics on shutdown due to a memory leak, casting the leaked address to `CBase*` will often give you the type of the leaked object.
4. An important recent addition to the functionality available for Symbian OS developers is on-target debugging. Whilst this is not currently available for all SDK and tool variants, most of the newest SDK and IDE releases do support this. Where available, you are advised to use it prior to releasing your application to help track down any phone-specific defects. See your SDK and/or IDE documentation for more information.