Essential Symbian OS

CODING TIPS

symbian
Press

Symbian Signed, developed in partnership between Symbian, its licensees, network operators and developers, promotes best practice to test and sign applications for Symbian smartphones.

Based on developer feedback, the Symbian Signed processes have been improved to make signing faster, easier and cheaper for developers driving growth in the number of Symbian applications.

- Open Signed - provides a free offline signing process using Developer Certificates allowing an application to be installed and tested on up to 1000 devices controlled by declared IMEIs (unique phone identification numbers).
- Express Signed - an entirely new process to allow software developers with Publisher IDs to instantly sign all applications that do not require access to the more sensitive device functions, allowing faster time-to-market. Developers can now sign applications on Expressed Signed for as little as US $20.
- Certified Signed - for those applications that require access to restricted system capabilities and/or where independent testing is required.

For more information visit www.symbiansigned.com.

Essential Symbian OS

CODING TIPS

from

symbian Press

Coding Tips
part of the Essential Symbian OS series

3rd edition: 12/07

Published by:
Symbian Software Limited
2-6 Boundary Row
Southwark
London SE1 8HP
UK
www.symbian.com

Trademarks, copyright, disclaimer
'Symbian', 'Symbian OS' and other associated Symbian marks are all trademarks of Symbian Software Ltd. Symbian acknowledges the trademark rights of all third parties referred to in this material. © Copyright Symbian Software Ltd 2008. All rights reserved. No part of this material may be reproduced without the express written permission of Symbian Software Ltd. Symbian Software Ltd makes no warranty or guarantee about the suitability or accuracy of the information contained in this document. The information contained in this document is for general information purposes only and should not be used or relied upon for any other purpose whatsoever.

Compiled by:
Phil Spencer
Hamish Willee

Managing Editors:
Satu McNabb
Ashlee Godwin

Design Consultant:
Annabel Cooke

Updated by:
Hamish Willee
Mark Shackman

Acknowledgements
Thanks to all the highly skilled and experienced software engineers who have shared their tips and best practice over the years. Many of the tips in this booklet are drawn from their collective wisdom.

Introduction

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

Contents

General Tips 2

Design Tips 4

Coding Tips 6

Testing Tips 21

Debugging Tips 22

References 25

Developer Resources 26

General Tips

1. The Symbian Developer Network hosts many valuable resources to help you write applications for Symbian OS. Visit *http://developer.symbian.com* regularly to get the latest SDKs, technical information, code examples and white papers. The Developer Network website provides access to:

    - The Symbian OS FAQ database *(http://developer.symbian.com/faqs)*, an invaluable source of information for developers, covering the most frequently asked design and coding questions.

    - Technical documentation and white papers, regularly published at *http://developer.symbian.com/main/oslibrary/*, and on the Symbian Developer wiki at *http://developer.symbian.com/wiki*.

    - Symbian Developer Library, the canonical API reference and guide for Symbian OS.

    - Downloads including: example applications; utility libraries; 'optional' system APIs, which are part of Symbian OS but may not ship on all phones; ported libraries that allow some APIs on newer Symbian OS versions to run on older OS versions.

- Information about tools for debugging and development.

- Newsgroup forums *(http://developer.symbian.com/faqs)* where you can get advice from Symbian engineers and the wider developer community. Forums exist to cover all aspects of development, from C++ and Java-specific questions, to hardware and to the Symbian Signing process.

2. Sign up to the Symbian Developer Network newsletter *(http://developer.symbian.com/register)* as part of your registration to the Symbian Developer Network website. This newsletter is distributed monthly by email and is the best way to keep up to date with the latest Symbian developer news.

3. Mobile phone manufacturers that use Symbian OS also run developer programs. You should register at their sites to get access to the latest phone-specific information and tips.

4. Get accredited! The Accredited Symbian Developer qualification *(http://developer.symbian.com/asd)* shows employers you are a competent Symbian OS engineer, and helps you to objectively assess your Symbian OS competencies.

5. Symbian Signed promotes best practice in designing applications to run on Symbian OS phones. Symbian Signed applications follow industry-agreed quality guidelines and support network operator requirements for signed applications. Go to the Symbian Signed website *(www.symbiansigned.com)* to learn more about the signing process.

6. There is a vibrant developer community outside of Symbian and our licensees. See *http://developer.symbian.com/main/getstarted/developer_community* for links to a number of prominent third party developer sites.

7. Finally, there are a growing number of Symbian OS publications available. Symbian Press *(http://developer.symbian.com/books)* offers books and booklets on a range of topics, all designed to help you write Symbian OS code more easily and reliably. Foreign language translations are now available for many of these!

Design Tips

8. 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 handle this modularization 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. An alternative approach is to make this distinction at source level so that your application builds as one single EXE file, but your 'engine' and UI-related code is separated into different CPP and H files for easier management and debugging.

9. Design with support for localization in mind. Never hard-code strings or literals that are intended to be viewed by application users into your source files – use the resource file mechanism Symbian OS provides to store strings.

10. Use only those APIs which are documented and supported for given SDK and Symbian OS releases. Symbian reserves the right to change or remove APIs which are not intended to be used by external developers in future releases. APIs tagged as deprecated should be avoided if possible; these are still supported in the current releases but have been marked as 'ready for removal' in a subsequent release.

11. Do not assume all 'system' files, for example, sound

files, images, etc. will be present on all phones. At the very least, you should always consider proper handling of the error condition should the file not be present on future handsets.

12. Do not assume that all Symbian OS DLLs will be present on all phones. Symbian OS licensees have a great deal of flexibility to customize their handsets; while some components are mandatory, others are optional *(http://developer.symbian.com/main/downloads/papers/SymbOS_cat/SymbianOS_cat.html)*. One method of working around optional DLLs is to create and (dynamically) load an ECOM plugin which in turn 'statically' loads the problem DLL. If the optional DLL is not present, the calling code can handle the `KErrNotFound` leave on `CreateImplementationL()`.

Coding Tips

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

13. 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 S60) in your `AppUi::HandleCommandL()` method.

14. 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 that you respond correctly when the user receives a high priority notification. For example, ensure that 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 act appropriately 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 that you're not doing anything which prevents the framework from operating as intended.

15. 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 to Symbian Signed for testing.

16. The stack is small! Where possible, allocate objects on the heap rather than on the stack. Unexplained KERN-EXEC 3 crashes are often symptomatic of stack corruption/overflow.

17. An application panic indicates a real bug in your code. Panic codes are documented in the Symbian Developer Library alongside each method, and in the System Panic Reference (**Symbian OS reference › System panic reference**). Here are some common errors:
    - Forgetting to add non-member, heap-allocated variables to the cleanup stack.
    - Putting member variables on the cleanup stack - never do this; just delete them in your destructor as normal.
    - The 'double delete'. For example, failure to correctly `Pop()` an already destroyed item from the cleanup stack, causing the stack to try and delete it again at a later time, or deleting a member variable when it has been used, but forgetting to set it to `NULL` so that it is deleted again in the destructor.
    - Accessing functions in variables which may not exist in your destructor. For example, the code below will panic if `iSomeServer` is `NULL` - which might happen if your object is destroyed before you allocate the memory, or if you have deleted the memory somewhere else in the application:

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

You should instead code this as:

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

- Calling a function on a NULL pointer
- Calling a function on a variable that has since gone out of scope, for example, by passing a stack-based variable into a callback to an asynchronous function.

18. Failing gracefully when system resources are unavailable is important. The most constrained resource is usually system RAM, so you need to be careful to handle low memory correctly. Two-phase construction and the use of the cleanup stack, as described below, are essential parts of such defensive programming.

19. 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);
```

You can also use `CleanupReleasePushL()` for "M" and "R" classes which use `Release()` instead of `Close()`. `CleanupDeletePushL()` calls `delete` on the pushed object, and `CleanupArrayDeletePushL()` calls `delete` on a pushed array.

20. In addition, remember that the cleanup stack 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 cleanup. See the Symbian OS Library documentation on `TCleanupItem` for more information.

21. If you have cause to use a `TRAP` of your own, do not ignore errors that you do not handle. 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 one above, leave for other errors:

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

22. Do not wait to `PushL()` things on to the cleanup stack. Any newly allocated object (except member variables) should be added to the cleanup 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::PushL(myObject1);
```

```
  CleanupStack::PushL(myObject2);
  // Do something more with the variables
  …
  CleanupStack::PopAndDestroy(2);
  // myObject2, myObject1
  }
```

because the allocation of `myObject2` could fail, leaving `myObject1` 'dangling' with no method of cleanup. 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
  }
```

23. 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.

24. 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 up 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?'

25. Use the `_LIT()` macro instead of `_L()`. The `_L()` has been deprecated since Symbian OS v5. 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`. You should also consider whether you should be using a hard coded literal at all; hard coded literal descriptors may need to be re-coded if you later localize your application.

26. Avoid declaring literals in your header files because all CPP files that include the header will generate a new copy, leading to code bloat. If there is a need for your `_LIT` to be visible to a number of files, put it in a CPP file and have a method that returns a reference to it.

27. When using descriptors as function parameters:
    - Use the base class by default as this allows clients of the function to use the concrete descriptor type that suits them best. For non-modifiable descriptors use `const TDesC&` and for modifiable descriptors use `TDes&`.
    - Always pass `TDesC` or `TDes` parameters by reference rather than by value. The rules of C++ mean that passing by value will create a copy of the `TDesC/TDes` (which has no data)

rather than the descriptor you intended.

- Always specify `const` for `TDesC` parameters. This avoids compiler errors when you pass literals into the function.

28. Do not instantiate `TDesC` or `TDes` as these don't provide storage space for data or for a pointer to data; instead you should declare the concrete classes derived from them.

29. Do not try to set the size of a stack descriptor as a run time-calculated value. Stack classes are templated and hence their size must be determined at compilation time – the following will result in a compilation error:

```
TInt length = someValue;
TBuf<length> myDescriptor;
```

30. Do not place large descriptors (or other objects) on the stack as it is a very limited resource. As a general rule, if a descriptor is greater than 256 bytes, consider allocating it on the heap. There are some stack descriptor `typedef`s which allocate more than this, and which therefore should be used with care: for example, `TEntry`, `TFileName`, `TFullName`, `TName`.

31. Do not allocate larger-than-necessary buffers on the off-chance that a large buffer will be required; this

is wasteful of memory! Instead allocate buffers that are closer to the normal size, and expand as necessary.

32. Be wary of the `TPtr` assignment operator (`TPtr::operator=()`). `TPtrs` come in two types, one which has a pointer to data (type 2), and one which contains a pointer to another descriptor (type 4). Attempting to assign the value of a type 2 with a type 4 `TPtr` will result in a panic (even though the reverse works). It is safer to use the `TPtr::Set()` method, as this changes the type of the descriptor appropriately.

33. Be wary that type 4 `TPtr` copy constructors do not behave in the same way as their assignment operators. The copy constructor creates a new descriptor that points to the same descriptor as the original, while the assignment operator copies the data from the first descriptor to where the second descriptor is pointing. For a type 2 `TPtr` , which has a pointer to data rather than another descriptor, the copy constructor and assignment operator behave the same so that assignment causes the second descriptor to point to the same data as the original.

34. Use an `RBuf` in preference to an `HBufC` for strings that you intend to manipulate/modify.

35. 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 an `HBufC` as an argument to a method which takes a `TDesC&` parameter (as recommended above). Note that, as you're modifying the string, you should consider using an `RBuf` instead of an `HBufC`.

36. 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 an `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.

37. Use `Alloc()` when creating an `HBufC` from a descriptor. A common mistake is to do:

```
HBufC* myData =
           HBufC::NewL(someBuffer.Length);
TPtr ptr= someBuffer.Des();
ptr.Copy(someBuffer);
```

when you can just do:

```
HBufC myData = someBuffer.AllocLC();
```

38. Be wary of `HBufC::ReAllocL()` and `HBufC::ReAlloc()` as these may return a different address from the original `HBufC*`. It is a common mistake to continue to use (invalid) `TPtr` pointers to the old buffer after re-allocation.

39. When using a method which takes a formatting specifier, use `%s` for formatting C style strings and `%S` for formatting descriptors. Formatting with the wrong type can raise an exception.

40. Initialize `TPtr` class data members in the constructor initialization list. `TPtr` does not have a default constructor so doing otherwise will result in a compilation error.

41. Perform bounds checking when appending or inserting data to your descriptor. Descriptors do this check internally, and will panic if your changes cause a descriptor to exceed its maximum length.

42. When passing or returning an object to a function, ensure that if you take ownership of the object you delete it! Symbian uses the convention that pointers in a method indicate transfer of ownership, while references indicate that caller retains ownership of the passed object.

43. Active objects and the active scheduler are key Symbian OS frameworks. 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:
    - The active scheduler places a `TRAP` around `RunL()` and, in the event of a leave, calls `CActive::RunError()`. You can choose to `TRAP` in the `RunL()` and continue operation, but it is usually easier and 'cleaner' to simply handle the error cases in your `RunError()`.
    - 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 destructor. There are cases where a service API cannot be cancelled, so your `DoCancel()` will be empty and calling `Cancel()` in the active object destructor will do nothing – but this will do no harm, and it's a good habit to develop.
    - Ensure that `DoCancel()` doesn't contain any cleanup code that must always be run – calling `Cancel()` will only result in a call to `DoCancel()` if the request is active.

44. 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 – see the 'Roids' technical paper on the Symbian Developer Network website *(http://developer.symbian.com/roidsgame)*.

45. 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://developer.symbian.com/faqs*.

46. 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.

47. Take care when implementing framework classes in your application. You should always derive from the platform-specific framework classes provided. For example, on UIQ, derive your Application UI class from `CQikAppUi` rather than from `CEikAppUi`. All of the UI-specific application base classes (`CQikAppUi`, `CQikApplication`, and `CQikDocument`) add functionality that the platform UI framework relies on to make applications perform correctly.

Testing Tips

48. 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.

49. 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://developer.symbian.com/faqs* also offers useful information.

50. 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.

Debugging Tips

51. Always debug on the emulator first; most problems that occur on both the emulator and hardware are much easier to debug on the emulator.

52. When writing and debugging a new `CCoeControl`-derived class, put `iEikonEnv->WsSession().SetAutoFlush(ETrue)` in the `ConstructL()` function of your Application UI. This means that graphics context (gc) draw commands will show up in the emulator immediately, rather than when the window server client buffer is next flushed. Edit your WSINI.INI file (\epoc32\release\winscw\udeb\system\data\) and ensure that the keyword FLICKERFREEREDRAW is not present. 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.

53. 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://developer.symbian.com/faqs* to download the tool and read further information.

54. If your application panics on shutdown due to a memory leak, casting the leaked address to `CBase*` on your IDE will often give you the type of the leaked object. The Hooklogger tool (which can be downloaded from the Symbian Developer Network) is often even more useful for debugging memory leaks.

55. 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 latest SDK and IDE releases do support this. Where available, this can be used to track down any phone-specific defects. See your SDK and/or

IDE documentation for more information.

56. Make sure 'Just in Time' debugging is enabled by:
    - Adding the following macro(s) to \epoc32\data\epoc.ini:
      'JustInTime debug', or 'JustInTime query'.
    - Ensuring that 'JustInTime 0' or 'JustInTime none' are removed from your epoc.ini file.
    - Calling `User::SetJustIntime()` within your code
    - If you're using CodeWarrior, setting the following registry value:

```
[ HKEY_LOCAL_MACHINE\ SOFTWARE/Microsoft\
     Windows NT\ CurrentVersion\ AeDebug]
   "UserDebuggerHot Key" = dword : 00000000
   "Debugger" = \"C;\\apps//Metrowerks//
     bin\\IDE.exe\" - p  %ld  - e  % l d"
   "Auto" = "0"
```

57. Review debug messages in %Temp%epocwind.out

58. Use the available developer tools including LeaveScan, HookLogger, Panix and D_EXC to debug memory leaks and other errors. These and many other useful tools can be downloaded from the Symbian Developer Network *(http://developer.symbian.com/main/tools)*.

59. FAQ-1344 shows how you can enable diagnostics to debug platform security `(KErrPermissionDenied)` violations - making it easy to identify missing capabilities.

References

Symbian Developer Network newsletter
http://developer.symbian.com/register

Symbian OS FAQ database
http://developer.symbian.com/faqs

Symbian Press
http://developer.symbian.com/books

"Roids" Games Writing paper
http://developer.symbian.com/roidsgame

Active Objects paper
http://developer.symbian.com/main/downloads/papers/CActiveAndFriends/CActiveAndFriends.pdf

Developer Resources

Symbian Developer Network
http://developer.symbian.com

Symbian Developer Network newsletter
http://developer.symbian.com/register

Symbian OS Tools Providers
http://developer.symbian.com/main/tools

Sony Ericsson Developer World
http://developer.sonyericsson.com

Forum Nokia
http://forum.nokia.com

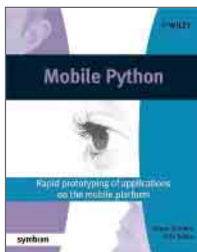
Sun Microsystems Developer Services
http://developer.java.sun.com

UIQ Developer Community
http://developer.uiq.com

New from

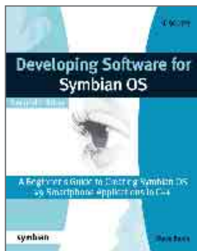
symbian
Press

Mobile Python



Mobile Python is a practical hands-on book that introduces the popular open source programming language Python to the mobile space. It teaches how to program your own powerful – and fun – applications easily on Nokia smartphones based on Symbian OS and the S60 platform.

Developing Software for Symbian OS, Second Edition



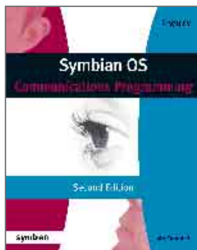
This second edition of *Developing Software for Symbian OS* helps software developers new to Symbian OS to create smartphone applications. The original book has been updated for Symbian OS v9 and now includes a new chapter on application signing and platform security, and updates throughout for Symbian OS v9 and changes to the development environment.

Symbian Press: *developer.symbian.com/press*

New from

symbian Press

Symbian OS Communications Programming, Second Edition



Targeting Symbian OS v9.1 and v9.2, *Symbian OS Communications Programming - Second Edition* will introduce you to the major communications functionality in Symbian OS and demonstrates how to perform common tasks in each area.

Symbian OS C++ for Mobile Phones, Volume 3

This book will help you to become an effective Symbian OS developer, and will give you a deep understanding of the fundamental principles upon which Symbian OS is based.

Symbian Press: *developer.symbian.com/press*

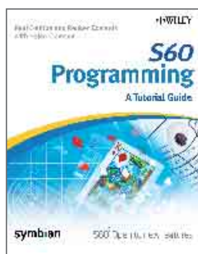
Also from

symbian
Press

The Symbian OS Architecture Sourcebook

This book conducts a rapid tour of the architecture of Symbian OS and provides an introduction to the key ideas of object orientation (OO) in software, with a detailed exploration of the architecture of Symbian OS.

S60 Programming



Fully up to date for Symbian OS v9 and S60 3rd Edition, S60 Programming is an essential foundation to developing software for Symbian OS.
This practical book is based on the authors' experiences in developing and teaching an academic course on Symbian software development.

Symbian Press: *developer.symbian.com/press*

Also from

symbian
Press

For all Symbian C++ developers:

Symbian OS C++ for Mobile Phones – Volume 1
by Richard Harrison

Symbian OS C++ for Mobile Phones – Volume 2
by Richard Harrison

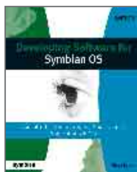
Symbian OS Explained
by Jo Stichbury

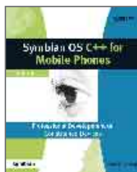
Symbian OS Internals
by Jane Sales

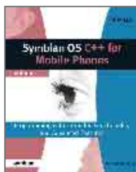
Symbian OS Platform Security
by Craig Heath

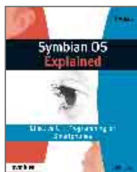
Smartphone Operating System Concepts with Symbian OS
by Mike Jipping

Accredited Symbian Developer Primer
by Jo Stichbury & Mark Jacobs









Also from

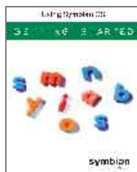
symbian Press

Published Booklets

Coding Tips
Performance Tips
Getting Started
Java ME on Symbian OS
P.I.P.S
Carbide.c ++
Data Sharing Tips
Essential S60 - Developers' Guide
Essential UIQ - Getting Started
Ready for ROM

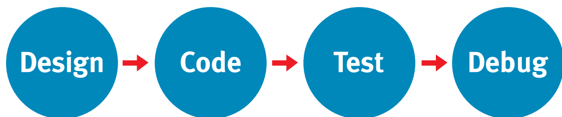
Translated booklets available in:

Chinese
Japanese
Korean
Spanish



Essential Symbian OS

CODING TIPS



Progressing through the stages of application production, from design to debugging, this mini-book provides advice on how to produce high-quality software for Symbian OS.

Coding Tips is part of the Essential Symbian OS series: a series designed to provide information in a handy format to Symbian OS developers.

Symbian Press
Symbian Press publishes books designed to communicate authoritative, timely, relevant and practical information about Symbian OS and related technologies. Information about the Symbian Press series can be found at **www.symbian.com/books**