Wiki

Coding Standards and Conventions

From Symbian Foundation

Contents

Introduction and General Principles

This document specifies general **Coding Standards and Conventions** for the Symbian Foundation when programming in Symbian OS C/C++. The document is owned and maintained by the foundation Architecture Council (http://developer.symbian.org/main/about/councils/council/architecture_council.php) .

Definitions

For definitions for the terms such as **Component**, **Collection**, **Package**, **Platform API** or **Public API** refer to the Glossary

The Purpose of This Document

The purpose of this document is to:

Define coding *recommendations and guidelines* for the Symbian Foundation developer community
Help package contributors to program according to the recommendations
Help package owners to verify contributions against the recommendations
Be the basis for extending and improving this document with new recommendations and examples to be created by the community
Refer to other relevant information sources that extend and support this document. For example, 'package guidelines'.

The purpose of this document is **not** to provide extensive instructions for Symbian OS C/C++ programming.

The document contains two different elements:

Recommendations, which have permanent numbers prefixed with **SFCSR** (Symbian Foundation Coding Standard Recommendation).
The target is that each recommendation is checkable by a tool.
Guidelines, which explain the recommendations, giving examples.

Motivation

Programming style can be defined as: *...the way that a programmer brings clarity, maintainability, testability, reliability, and efficiency to the code of a module.*

This definition sets the objectives for good programming style but it does not help to determine whether a piece of software is of good or bad style. One can easily conclude that coding itself is only one factor in reaching these goals: a badly designed module can hardly be clear, maintainable, testable, or reliable, even if it is coded in the best possible style. On the other hand, even the most excellent design cannot make the software clear and maintainable if its implementation, the code, is of bad quality.

In Symbian Foundation code it is necessary that the programmers have a similar style. Otherwise software maintenance is too expensive and difficult and re-use becomes almost impossible. A uniform way of coding contributes strongly to the programming productivity of the community.

General Principles

When requiring clear and maintainable code, certain general recommendations can be applied to all situations:

Remember **re-usability**. Re-usability means utilizing existing package external APIs, not code copying.
Check first whether there is a ready-made package external API that offers the constant, macro or class that you need. This will help you avoid hard-coding constants, and it will help protect your code against system-wide changes.
If you see a need for a new common component, communicate the need to community and package owners.
Do not hard-code directory references or make any assumptions about the source tree location of your component.
Remember **maintainability**.
One of the properties of good code is that it is understandable and easy to read. If/when the code needs to be maintained, it can be a very difficult and time-consuming job if the maintainer does not understand what the code does. Thus, when writing code, you should always keep asking yourself how others can understand it.
Write **in-source documentation** (comments) to describe the API. These comments are very valuable for users of the API to understand the purpose of the API, what steps are required to handle errors and return values, and so on. A good approach is to include and example usage block into each header file of the API.
Write comments whenever it would help in understanding implementation code. Code comments are essential for making code easier to read and understand, not only for a later maintainer but also for the coder himself, who will soon forget the assumptions he used when writing the code.
Do not make assumptions about the user of your code. Make the code protect itself against misuse. If this is not possible, remember to document all restrictions and especially error codes returned by functions. (This does not apply to all standard Symbian OS error codes; use common sense.)
Code should pass **compilation without warnings**. There must be a well-known and very good reason to ignore warnings, and the developer must know why the warning is there. Warnings that should genuinely be suppressed throughout the system should be suppressed by a common header file.
Use code-analysis tools, such as CodeScanner, PC-Lint or a similar source code checker, to make additional checks on code to catch some of its bugs, inconsistencies, and oversights.
Remember **internationalization** issues. Internationalization means designing and coding software to allow for localization and for smooth production of language variants, minimizing engineering changes. Internationalization aims at a single code base.
Make components **configurable**. See the Configurability Guidelines.

Notes

Please note that the extant offering from the foundation contains deviations from the coding standards. These standards, however, are strongly recommended for new contributions as well as for the old source code. We'll try to gradually fix the existing deviations.

Naming Conventions

General Naming Guidelines

SFCSR-100 Naming in General

All names should be in English (US), because the reserved words of the programming language are in English, too. In addition, in an international organization, English is usually the only language everybody understands.
Always use meaningful and carefully considered names.
Avoid using names that can easily be confused or which differ only in case.
Use descriptive names for identifiers. If you cannot name an identifier, consider whether it has the right to exist at all.
If you use an abbreviation in an identifier name, capitalize it as a regular word, for clarity. For instance: not `CSIPProfile`, but instead `CSipProfile`.
Function, **class**, **typedef** and **struct** names begin with an uppercase letter.

What Makes Up a Good Name?

Use full English descriptors that accurately describe the variable/field/class/. For example:
Use names like *firstName*, *grandTotal*, or *CorporateCustomer*. Although names such as *x1*, *y1*, or *fn* are easy to type because they're short, they do not provide any indication of what they represent and result in code that is difficult to understand, maintain, and enhance (Nagler, 1995; Ambler, 1998a).
Use terminology applicable to the domain.
If your users refer to their clients as customers, then use the term *Customer* for the class, not *Client*. Many developers will make the mistake of creating generic terms for concepts when perfectly good terms already exist in the industry/domain.
Use mixed case to make names readable.
You should use lower case letters in general, but capitalize the first letter of class names and interface names, as well as the first letter of any non-initial word (Kanerva, 1997).
Avoid abbreviations. If you need to do so, e.g. for shortening a name, then use them intelligently.
This means you should maintain a list of standard short forms (abbreviations), you should choose them wisely, and you should use them consistently.
For example, if you want to use a short form for the word number, then choose one of *nbr*, *no*, or *num*, document which one you chose (it doesn't really matter which one), and use only that one.
Capitalize the first letter of standard acronyms .
Names will often contain standard abbreviations, such as *SQL* for Standard Query Language. Names such as sqlDatabase for an attribute, or SqlDatabase for a class, are easier to read than sQLDatabase and SQLDatabase.

SFCSR-101 Camel Casing

The name of a class, function, constant or variable must be written in CamelCase with an appropriate prefix or suffix. Write all words adjoined in CamelCase, for example LikeThis.

Examples:

Right:

```
void TObject::PrepareForCommit(); // all three words adjoined and capitalized
class CGlobalText;

TInt CTetrisClass::Function() // member function
    {
    TInt automaticVariable( KInitialValue ); // integer variable
    TBool status( IsItTrue() ); // Boolean variable
    iMemberObject.Function(); // calls a function
    // ... more
    }
```

Wrong:

```
TBool Variable = is_it_true();
// Name of an automatic variable mistyped,
// and incorrect function name.
```

SFCSR-102 Disallowed Characters

Avoid numbers in identifiers. Never use the underscore (_). The only exception is its use as a delimiter in pre-processor macro names. See Macros.

SFCSR-103 Disallowed Words

Do not use company internal names, such as product, platform, project or company names.

Do use your own component-specific identifier to prefix the identifiers of your project, after the possible one-letter Symbian prefix.

Do not use offensive words or profanities.

File and Directory

SFCSR-110 Disallowed Characters in Directory and File Names

Do not use white space or non-ASCII character in directory or file names.
Do not use the following characters:

```
& ^ + - @ $ % * ( ) |  / [ ] { } < > ? ; : , " '
```

Names containing spaces or other special characters require special processing, especially in command lines.

SFCSR-111 Directory and File Names Should be Lowercase

The name of a directory in the source tree should be in lowercase.
The name of a file in the source tree should be in lowercase.
A reference to a directory or a file, for example, `#include` statement, should be in lowercase.

SFCSR-112 Maximum Length of Qualified File Name

The maximum length of a qualified file name, that is a file name **including** package, collection and component name, is 140 characters. The limitation has been set to avoid problems during the build process.

Examples:

```
/sf/mw/serviceapifw/rtsecuritymanager/inc/rtsecmgrdef.h
/sf/os/devicesrv/commonservices/commonengine/src/commoncontentpolicy.cpp
```

The **bold** name length shall be no more than 140 characters long.

If you need to shorten directory or file names, you can use abbreviations, such as:

application inter-working → `aiw`
connectivity → `conn`
configuration → `config`
control → `ctrl`
definition → `def`
declaration → `decl`
developer → `dev`
documentation → `doc`
framework → `fw`
hardware → `hw`
language inter-working → `liw`
library → `lib`
localization → `loc`
management → `mgmt`
manager → `mgr`
multimedia → `mm`
runtime → `rt`
platform → `plat`
public → `pub`
security → `sec`
server → `srv`
service → `srv`
service api → `sapi`
software → `sw`
source → `src`
system → `sys`
utility → `util`
utilities → `utils`

SFCSR-113 Forward Slash

A reference to a directory or a file shall use only forward slashes (/).

This is to ensure compatibility with Unix-based build systems.

SFCSR-114 No Hardcoded Directory Names

Do not use a hardcoded reference to a directory because:

In devices, the runtime environment can be different.
In build environment, you need the flexibility and freedom to make changes without major changes to the software.

Use available macros and APIs to retrieve:

correct **SYSTEMINCLUDE** path in `.mmp` file (declared in the `platform_paths.hrh`)
correct **RESOURCE** path in `.mmp` file (declared in `data_caging_paths.hrh`)
correct **EXPORT** path for platform or public API header (declared in the `platform_paths.hrh`)
correct **TARGETPATH** path for component resources (declared in the `platform_paths.hrh`)
correct name of device or MMC root directory (declared in the `pathconfiguration.hrh`)
correct directory for a a media file at run-time (declared in the `pathinfo.h`)

Keep source files of a package source tree independent and free from any assumptions of file locations. **Examples:**

```
// In application layer .mmp file use APP_LAYER_SYSTEMINCLUDE
APP_LAYER_SYSTEMINCLUDE

// In .mmp file use RESOURCE_FILES_DIR
TARGETPATH RESOURCE_FILES_DIR

// In bld.inf use MW_LAYER_PUBLIC_EXPORT_PATH
PRJ_EXPORTS
../inc/AiwCommon.hrh MW_LAYER_PUBLIC_EXPORT_PATH(AiwCommon.hrh)

// In .cpp file, get the root path of Phone Memory at run-time.
#include <pathinfo.h>
TFileName path = PathInfo::PhoneMemoryRootPath();
```

SFCSR-115 Name File After the Declaration

Naming files after the main declarations they contain makes maintenance easier. Many header files will contain more than one declaration, for example, C class and related M class.

Examples:

File `aiwservicehandler.h` contains class `CAiwServiceHandler` declaration.
File `aiwservicehandler.cpp` contains class `CAiwServiceHandler` implementation wrapper.
File `aiwservicehandlerimpl.cpp` contains class `CAiwServiceHandlerImp` actual implementation class.

Exceptions

Third-party code.
Code generated by third-party tools such as Rational Rose.
If mandated by another standard that also applies to that code.

Class and Variable

The class naming convention reinforces essential programming idioms.

SFCSR-120 Class Names

Class names consist of <**Symbian OS prefix**><**Optional component prefix**>**Name**.

The length of a component prefix is two to six letters.

The **Symbian OS prefix** should be one of the following:

C for heap-allocated classes derived from a base class CBase.
R for resource classes containing handles to a real resource which is maintained elsewhere.
T for classes which do not own heap-allocated data, thus do not need a destructor, and thus can be safely allocated on the stack.
M for interface classes, which contain only pure virtual functions and no member data. This is the only class type from which multiple inheritance is allowed.
D for kernel-side heap-allocated classes derived from a base class DBase.

SFCSR-121 Static Classes

Static classes have no prefix letter. That is, static class name should not start with **C**, **R**, **T**, **M** or **D** to prevent mixups

SFCSR-122 UIKON Classes

For UIKON classes, the **component prefix** is **Eik**. For AVKON (S60 extension of UIKON), it is **Akn**.

Examples:

```
class CBase;
class TTypefaceInfo;
class RFont;
class MLaydoc;
class User; // static class
class EikUtils; // static class
class Time; // static class
class MTetrisObserver; // observer class (mix-in)

/**
* Tetris game factory class.
*/
class TetrisFactory // static class
    {
public:
    /** Constructor */
    static CTetrisItem* NewItemL();
    ....
```

SFCSR-123 Classes With X Prefix

An X class is recommended when a class deviates from the recommendations specified elsewhere in this coding standard.

An X class should be considered if the best solution to a problem involves a class that doesn't quite fit the recommendations, but, because of the way in which the class will be used, the clients of the class can cope with its unusual recommendations.
Users of an X class should carefully read and follow the comments that explain how it differs from standard Symbian classes.

Examples:

An 'almost-R' class that has a close method but can't be arbitrarily bitwise copied.
An 'almost-M' class that has data members (so some forms of multiple inheritance are to be avoided).
An 'almost-T' class that has a non-trivial destructor.

SFCSR-124 Member Variables

The name of a member variable must be prefixed with an **i**.

Examples:

```
class TObject
    {
    ...
private:
    TType iType;
    TInt iElementOffset;
    TPtrC iComponentValue;
    };

/**
* Tetris game base class.
*/
class CTetrisClass: public CBase
    {
    ...
private:
    TInt iIntegerMember;
    COtherClass* iAnotherObject;
    ...
```

SFCSR-125 Variable Name Length

Avoid variable names longer than 15 characters.

SFCSR-126 Automatic Variables

The first letter of an automatic (local) variable must be in lower case. The remainder of the name must be in **lower camel case**, i.e. lowerCamelCase.

Making the first letter of an automatic variable lowercase is simply a convention to help reviewers and maintainers to follow your code.
In the name, make clear that name is not member variable or parameter.

Examples:

```
TInt elementOffset;
TInt index;
TInt altitude;
```

SFCSR-127 Global Variables

The use of global variables is discouraged. But, if you cannot avoid using a global variable its name must have a small **g** prefix.

Macros

SFCSR-130 Macro Names

A macro name must be in UPPER_CASE with underscores joining words.
Macros names created by Symbian Foundation should start with 'SYMBIAN_'. Older macros started with two underscores ('__'). Do not use these in your own macro names.

Examples

```
// macro names
IMPORT_C
__ASSERT_ALWAYS

// Macro declaration in Symbian_OS.hrh
#define SYMBIAN_SECURE_ECOM
```

Exceptions

Header file guards to protect against multiple inclusions start with just the name of the header file. See here for more details.

Functions

As mentioned in SFCSR-100, function names begin with an uppercase letter.

SFCSR-140 Set and Get Functions

A setter method must be named **Set**Xxx().
A getter method must be named
Xxx() if it returns the member item or value, and
GetXxx() if it uses a reference parameter to supply the item or value.
Use **const** attribute to indicate that a function does not change the state of an object.

Examples:

```
// Declarations
void SetOffset( TInt aOffset ); // setter
TInt Offset() const; // getter
TInt GetCharFormat( TCharFormat& aFormat ) const; // getter

// Usage
TInt offset = Offset();
TCharFormat format;
TInt err = GetCharFormat( format );
```

SFCSR-141 Leaving Functions

The name of a leaving function must have an **L** suffix. For proper error handling, it is very important for the caller to know whether a function might leave or not.

SFCSR-142 Function Pushes Item to Cleanup Stack

The name of a function that delegates responsibility for removing **one** item from the cleanup stack must have a **C** suffix. The item must be popped later from the *cleanup stack* by the caller.

SFCSR-143 Function Destroying Object

The name of a function that destroys the object on which it is called must have a **D** suffix.

SFCSR-144 Order of Function Suffix

If more than one of the three recommendations above applies to a function, the suffixes must appear in the order above (L,C,D)

Examples:

```
CStoreMap* map = CStoreMap::NewL(); // Leaving function
CStoreMap* map = CStoreMap::NewLC(); // map remains on cleanup stack
CEikDialog* dialog = new ( ELeave ) CBossSettingsDialog;
if ( dialog->ExecuteLD( R_BOSS_SETTINGS_DIALOG ) ) // dialog is deleted
```

SFCSR-145 Function Parameters

Parameter names must be mentioned in declaration and definition.
If some parameter is not referenced inside the function, its name must be commented out in the definition, to avoid compilation warnings.
The name of a function parameter must be prefixed with an **a**.
Note, that if the name begins with a vowel, the prefix is NOT 'an'.
Passing weak parameter types (TAny* style) should be avoided. Use discrete types whenever possible.
Use a **const** attribute to indicate that a parameter is read-only.

Examples:

```
void TObject::TObject( TType aType, TInt aElementOffset )
    {
    iType = aType;
    iElementOffset = aElementOffset;
    }
const CHugeMatrix& Matrix() const; // returns read-only reference
```

Other Function-Related Guidelines

For recommendations related to the use of functions and member functions, please see the article on Objects and Classes.

Pointers, Constants and Enumerations

SFCRS-150 Pointer and Reference Types

Place the pointer ***** or reference **&** specifier next to the type, not next to the name.

Examples:

```
TText* dataPointer;
void TDemo::Append( const TDesC& aData );
TEntry* TDemo::Entry() const;
```

SFCSR-151 Constants

The name of a constant must be prefixed with a **K**.

Examples:

```
const TInt KMaxNameLength = 0x20;
const TUid KEditableTextUid = { 268435548 }
_LIT( KTxtRootTitle, "Root" ); // Non-localizeable string
```

SFCSR-152 Enumerated Types

An enumeration name must be given a T prefix and named as a class.
The name of an enumerated value must be prefixed with an E.
An enumeration must be declared with the minimum necessary scope.

Try to avoid polluting the global name space.

Examples:

```
class TDemo
    {
public:
    enum TShape { EShapeRound, EShapeSquare };
    enum TFruit
        { // Fruit definitions
        EFruitOrange,
        EFruitBanana,
        EFruitApple
        };
    };
TDemo::TShape shape = TDemo::EShapeSquare;
```

Filetypes and Templates

Filetypes

The table below lists the file extensions for various source files:

Extension	Description	Related page
.inf (bld.inf only)	Main build info file for a package, collection or component.	See here
.bmp	Bitmap file	
.c	C source file	See here
.confml	Configuration mark-up language file. XML-based language used to describe configuration elements. Created by tool Configuration Tools.	
.cpp	C++ source file	See here
.crml	Central repository mark-up language file. Created using Configuration Tools	
.def	Module definition file, which lists the exported functions of a DLL. Generated by build tools.	
.dosc	Doxygen out-of-source comment file	
.h	C++ header file	See here
.hrh	Common header file for resources and C++ programs	See here
.iby	Contains a list of files that are included in the ROM image	See here
.hby	Header file for .iby files	See here
.inl	C++ inline function definition file	See here
.loc	Localization string resource header file	See here
.metaxml	XML formatted meta-data file for API project	See here
.mk	Extension makefile	See here
.mmp	Symbian OS project-specification file (for makmake)	See here
.mmh	Header file for .mmp files	See here
.pkg	Package file, input for creating SIS/SISX installation files.	See here
.rh	Resource header file	See here
.rss	Resource file	See here
.txt	Text file, for example Central Repository file or iconlist file.	
.xml	XML formatted file, for example backup_registration.xml	

When ordering files in typical order of creation:

bld.inf file
MMP file
H file
INL file
HRH file
RH file
LOC file
RSS file
CPP file

Build Files

bld.inf File

Main build info file for a package, collection or component.

SFCSR-210 Exporting Under the /epoc32/include

Export macros defined in the **platform_paths.hrh** shall be used in bld.inf to export files to /epoc32/include and subdirectories under it. This way

harmonizes the locations and order of include paths
gives future possibility to modify those directories
makes layering and API categorization visible in the include-hierarchy structure, thus making it easier to detect the allowed APIs that can be used.

To export APIs, use the bld.inf of the API directory. APIs should not be exported directly from the bld.inf of a package, collection or component.

The layer-specific export paths are defined by the macros in the **platform_paths.hrh** file
For exporting the public APIs, use the macros ending to _PUBLIC_EXPORT_PATH.
For exporting the platform APIs, use the macros ending to _PLATFORM_EXPORT_PATH.

API export example:

```
#include <platform_paths.hrh>

PRJ_PLATFORMS
DEFAULT

PRJ_EXPORTS

../inc/eikslbd.h     MW_LAYER_PLATFORM_EXPORT_PATH(eikslbd.h)
../inc/aknshortcuts.h     MW_LAYER_PLATFORM_EXPORT_PATH(aknshortcuts.h)
../inc/aknjavalists.h     MW_LAYER_PLATFORM_EXPORT_PATH(aknjavalists.h)
```

To export other files, use the other layer-specific macros:

For .iby files, use macros ending with _IBY_EXPORT_PATH.
For .loc files use the macros ending with _LOC_EXPORT_PATH.

For `.confml` files use macros ending with `_CONFML`.
For `.crml` files use macros ending with `_CRML`.

Example:

```
#include <platform_paths.hrh>

PRJ_EXPORTS

// IBY-files
../rom/avkon.iby CORE_MW_LAYER_IBY_EXPORT_PATH(avkon.iby)
../rom/aknmemcardui.iby CORE_MW_LAYER_IBY_EXPORT_PATH(aknmemcardui.iby)
../rom/avkonResources.iby LANGUAGE_MW_LAYER_IBY_EXPORT_PATH(avkonresources.iby)

// Generic configuration interface for component cenrep settings
../conf/s60/avkon.confml                APP_LAYER_CONFML(avkon.confml)
../conf/apac/avkon_apac.confml          CONFML_EXPORT_PATH(avkon_apac.confml,apac)
../conf/japan/avkon_japan.confml        CONFML_EXPORT_PATH(avkon_japan.confml,japan)
../conf/s60/avkon_101F876E.crml         APP_LAYER_CRML(avkon_101F876E.crml)
../conf/s60/avkon_102858F2.crml         APP_LAYER_CRML(avkon_102858F2.crml)
```

SFCSR-211 `bld.inf` Shall Use Extension for Scalable Application Icons

This is the way to include scalable application icons. In first place, all application icons shall be scalable.

Example:

```
// Use extension for scalable application icons
PRJ_EXTENSIONS
START EXTENSION s60/mifconv
OPTION TARGETFILE ImageEditorUi.mif
OPTION HEADERFILE ImageEditorUi.mbg
OPTION SOURCEFILE iconlist.txt
END
```

`bld.inf` Template

```
/*
* Copyright (c) {Year(s)} {Company}.
* All rights reserved.
* This component and the accompanying materials are made available
* under the terms of the License "Symbian Foundation License v1.0"
* which accompanies this distribution, and is available
* at the URL "http://www.symbianfoundation.org/legal/sfl-v10.html".
*
* Initial Contributors:
* {Name} {Company} - Initial contribution
*
* Contributors:
* {Name} {Company} - {Description of contribution}
*
* Description:
* {Description of the file}
*
*/
```

```
// Use standard macros
#include <platform_paths.hrh>

PRJ_PLATFORMS
DEFAULT

PRJ_EXPORTS

// Use extension for scalable application icons
PRJ_EXTENSIONS
START EXTENSION s60/mifconv
OPTION TARGETFILE ?name_aif.mif
OPTION SOURCES -c8,8 qgn_?name.svg
END


PRJ_MMPFILES

PRJ_TESTMMPFILES

PRJ_TESTEXPORTS
```

MMP File

Symbian OS project specification file (for makmake).

SFCSR-212 MMP File Shall Use Standard Macros

The MMP file

Shall use one, layer-specific `SYSTEMINCLUDE` macro (declared in the `platform_paths.hrh`). The upper-layer macro contains lower-layer paths, that is, lower-layer paths are a subset of the upper-layer paths.
Shall use macros to state from where the resource files are put (declared in the `platform_paths.hrh`).

The following benefits are achieved:

All the default `SYSTEMINCLUDE` paths are defined in a centralized place and the developer only needs to add one macro.
Possible additions/changes/removals can be done later on from a centralized place without the need to edit every file separately (so avoiding the need for every developer to edit their own files).
The order of the system directories is always the same.

SFCSR-213 `DEFFILE` Keyword Should Not Be Used in an MMP file

The default name (the same as the TARGET) and location of the DEF file should be used. It is more consistent to have the same name for the MMP file, the DEF file and the target.

This policy also avoids a maintenance overhead if a new DEF file format is added.

Exceptions: A DEFFILE statement may be needed if you are creating a polymorphic DLL that has to follow a specific interface defined elsewhere, or if you have to resolve compatibility issues between different versions of that DLL.

However, in general, this should be avoided and the polymorphic interface should have a specific target type described in the `TARGETTYPE` statement.

MMP File Template

Template for application layer MMP file

```
/*
* Copyright (c) {Year(s)} {Company}.
* All rights reserved.
* This component and the accompanying materials are made available
* under the terms of the License "Symbian Foundation License v1.0"
* which accompanies this distribution, and is available
* at the URL "http://www.symbianfoundation.org/legal/sfl-v10.html".
*
* Initial Contributors:
* {Name} {Company} - Initial contribution
*
* Contributors:
* {Name} {Company} - {Description of contribution}
*
* Description:
* {Description of the file}
*
*/
```

```
// Use standard macros
#include <platform_paths.hrh>
#include <data_caging_paths.hrh>
// Build target
TARGET name.exe|dll|dll
TARGETTYPE exe|dll|plugin
UID 0x100039CE|0x1000008D|0x10009D8D allocated-uid
// Decide what capabilities are needed
CAPABILITY NONE
// Use correct layer specific systeminclude macro
APP_LAYER_SYSTEMINCLUDE
// Define the vendor ID
VENDORID 0
START RESOURCE name.rss
HEADER
// Use correct macro for resource path
TARGETPATH APP_RESOURCE_DIR
LANGUAGE_IDS
END // RESOURCE
```

MMH File

The `.mmh` files are header files for `.mmp` files. These files can be included in the `.mmp` file using the `#include` directive.

Both the `.mmh` and `.mmp` files have the same format.

MMH files can be used to group the common contents from multiple `.mmp` files.
They can also be used for macro definitions in different build configurations.

IBY File

The `.iby` file is an image-description file. It is used for declaring what files will be included in the ROM image.

IBY File Template
SFCSR-214 IBY File Uses Standard Macros

Use macros declared in the `data_caging_paths_for_iby.hrh`

```
/*
* Copyright (c) {Year(s)} {Company}.
* All rights reserved.
* This component and the accompanying materials are made available
* under the terms of the License "Symbian Foundation License v1.0"
* which accompanies this distribution, and is available
* at the URL "http://www.symbianfoundation.org/legal/sfl-v10.html".
*
* Initial Contributors:
* {Name} {Company} - Initial contribution
*
* Contributors:
* {Name} {Company} - {Description of contribution}
*
* Description:
* {Description of the file}
*
*/
```

```
#ifndef FILENAME_IBY
#define FILENAME_IBY
// Use standard macros
#include <data_caging_paths_for_iby.hrh>
#endif
```

HBY file

The `.hby` files are header files for `.iby` files. They can be used to declare platform specific macros. These files can be included in the `.iby` file using the #include directive.

Apply IBY file template.

PKG File

Package file, input for creating SIS/SISX installation files.

Normal PKG File Template

```
;
; Copyright (c) {Year(s)} {Company}.
; All rights reserved.
; This component and the accompanying materials are made available
; under the terms of the License "Symbian Foundation License v1.0"
; which accompanies this distribution, and is available
; at the URL "http://www.symbianfoundation.org/legal/sfl-v10.html".
;
; Initial Contributors:
; {Name} {Company} - Initial contribution
;
; Contributors:
; {Name} {Company} - {Description of contribution}
;
; Description:
; {Description of the file}
;
```

```
; Languages
&EN, FR, GE
;01, 02, 03

; Package header
#{"MyApp-EN", "MyApp-FR", "MyApp-GE"}, (0x12345678), 1, 2, 3, TYPE=SA

; Localised Vendor name
%{"Vendor-EN", "Vendor-FR", "Vendor-GE"}

; Unique Vendor name
:"Vendor"

; Platform Dependency
[0x101f796],0 ,0 ,0,{"S60ProductID", "S60ProductID", "S60ProductID"}

;; Platform UIDs:
;; S60 3.0 0x101f7961, S60 3.1 0x102032be, S60 3.2.x 0x102752ae, S60 5.0 0x1028315f

; -------- files --------

;; application files
"\epoc32\release\armv5\urel\myapp.exe"                -  "!:\sys\bin\myapp.exe"
"\epoc32\data\z\private\10003a3f\apps\myapp_reg.rsc"   -  "!:\private\10003a3f\import\apps\myapp_reg.rsc"
"\epoc32\data\z\resource\apps\myapp.mif"              -  "!:\resource\apps\myapp.mif"

;; Other binaries
"\epoc32\release\thumb\urel\myapp_engine_1.dll"-"!:\sys\bin\myapp_engine_1.dll"
"\epoc32\release\thumb\urel\myapp_engine_2.dll"-"!:\sys\bin\myapp_engine_2.dll"


;; language-dependent resource files

"\epoc32\data\z\resource\apps\myapp.r00"              - "!:\resource\apps\myapp.rsc"
"\epoc32\data\z\resource\apps\myapp.r01"              - "!:\resource\apps\myapp.r01"
"\epoc32\data\z\resource\apps\myapp.r02"              - "!:\resource\apps\myapp.r02"
"\epoc32\data\z\resource\apps\myapp.r03"              - "!:\resource\apps\myapp.r03"

; Alternative way for installing language dependent resource files
{
"\epoc32\data\z\resource\apps\myapp.r00"
"\epoc32\data\z\resource\apps\myapp.r01"
"\epoc32\data\z\resource\apps\myapp.r02"
"\epoc32\data\z\resource\apps\myapp.r03"
}   - "!:\resource\apps\myapp.rsc"
```

Stub PKG File Template

```
;
; Copyright (c) {Year(s)} {Company}.
; All rights reserved.
; This component and the accompanying materials are made available
; under the terms of the License "Symbian Foundation License v1.0"
; which accompanies this distribution, and is available
; at the URL "http://www.symbianfoundation.org/legal/sfl-v10.html".
;
; Initial Contributors:
; {Name} {Company} - Initial contribution
;
; Contributors:
; {Name} {Company} - {Description of contribution}
;
; Description:
; {Description of the file}
;
```

```
; Languages
&EN

; Header
#{"MyApp-EN"}, (0x12345678), 1, 2, 3, TYPE=SA

; Localised Vendor name
%{"Vendor-EN"}

; Unique Vendor name
:"Vendor"

; Files
;; Binaries
""-"z:\sys\bin\myapp.exe"
""-"z:\sys\bin\mylib.dll"

; Other files
;; wild character usage
;; '*' : multiple character matching
;; '?' : single character matching
""-"z:\resource\apps\myapp.r*"
""-"z:\resource\apps\myapp.h*"
""-"z:\private\22334455\*.cfg"
""-"z:\private\22334455\file.tx?"
```

ROM Update PKG File Template

```
;
; Copyright (c) {Year(s)} {Company}.
; All rights reserved.
; This component and the accompanying materials are made available
; under the terms of the License "Symbian Foundation License v1.0"
; which accompanies this distribution, and is available
; at the URL "http://www.symbianfoundation.org/legal/sfl-v10.html".
;
; Initial Contributors:
; {Name} {Company} - Initial contribution
;
; Contributors:
; {Name} {Company} - {Description of contribution}
;
; Description:
; {Description of the file}
;
```

```

; Languages
&EN, FR, GE
;01, 02, 03

; Package header
#{"MyApp-EN", "MyApp-FR", "MyApp-GE"}, (0x12345678), 1, 2, 3, TYPE=SA, RU

; Localised Vendor name
%{"Vendor-EN", "Vendor-FR", "Vendor-GE"}

; Unique Vendor name
:"Vendor"

; Platform Dependency
[0x101f796],0 ,0 ,0,{"S60ProductID", "S60ProductID", "S60ProductID"}

;; Platform UIDs:
;; S60 3.0 0x101f7961, S60 3.1 0x102032be, S60 3.2.x 0x102752ae, S60 5.0 0x1028315f

; -------- files --------

;; application files
"\epoc32\release\armv5\urel\myapp.exe"                -  "!:\sys\bin\myapp.exe"
"\epoc32\data\z\private\10003a3f\apps\myapp_reg.rsc"   -  "!:\private\10003a3f\import\apps\myapp_reg.rsc"
"\epoc32\data\z\resource\apps\myapp.mif"              -  "!:\resource\apps\myapp.mif"

;; Other binaries
"\epoc32\release\thumb\urel\myapp_engine_1.dll"-"!:\sys\bin\myapp_engine_1.dll"
"\epoc32\release\thumb\urel\myapp_engine_2.dll"-"!:\sys\bin\myapp_engine_2.dll"


;; language-dependent resource files

"\epoc32\data\z\resource\apps\myapp.r00"              - "!:\resource\apps\myapp.rsc"
"\epoc32\data\z\resource\apps\myapp.r01"              - "!:\resource\apps\myapp.r01"
"\epoc32\data\z\resource\apps\myapp.r02"              - "!:\resource\apps\myapp.r02"
"\epoc32\data\z\resource\apps\myapp.r03"              - "!:\resource\apps\myapp.r03"

; Alternative way for installing language dependent resource files
{
"\epoc32\data\z\resource\apps\myapp.r00"
"\epoc32\data\z\resource\apps\myapp.r01"
"\epoc32\data\z\resource\apps\myapp.r02"
"\epoc32\data\z\resource\apps\myapp.r03"
}   - "!:\resource\apps\myapp.rsc"
```

Header Files

H File

C/C++ header file.

SFCSR-220 Use Header Templates

Use the H file template to start programming a header.
The template brings correct order to declarations, for example:

Place class constructors and destructors at the beginning.
Place class data members at the end.
Always use separate sections for derived functions and add comments to describe from where they have been inherited.
Note
Header files should not contain code but only prototypes and declarations. So do not implement inline functions in a header file. Put them into separate **INL** files.

SFCSR-221 Class Declaration Order

A class declaration has the following order:

friend classes
functions (public, protected, private)
data (public, protected, private)

SFCSR-222 Header File Comments

Class and function comments (including parameters and return values) go in the header file. Comments are adequate if they:

Obey the general commenting recommendations.
Use the commenting style of the H file template.
Permit component test cases to be drawn up based on the comments.

The comments in an H file are part of other comments meant to be handled by the program **Doxygen**, which is an automatic class documentation generator.

There should not be any Doxygen errors or warnings when the header is processed.

SFCSR-223 Protect Against Multiple Includes

Add

```
#ifndef FILENAME_H
#define FILENAME_H
```

statements in the beginning of a header to protect it against multiple includes from other files.

Example:

```
// File featdiscovery.h
#ifndef FEATUREDISCOVERY_H
#define FEATUREDISCOVERY_H

// Declarations

#endif
```

Alternative style:

```
#if !defined(FEATUREDISCOVERY_H)
#define FEATUREDISCOVERY_H

// Declarations

#endif
```

SFCSR-224 Include Only Necessary Files

Inside a header file, include only those other header files that are required for class definition or to use the class. Do not use `#ifndef` statements when including other header files.

SFCSR-225 Exclude Implementation-Specific Declarations

Isolate implementation-only specific declarations (functions, types, constants, and so on) into separate internal header files. You can not include the internal header file from the header par of a public API or a platform API

SFCSR-226 Use Forward Declarations

If the class is accessed only via pointers or references, it should **not** be included with `#include`. Including unnecessary header files may very easily increase the build time and object file size significantly. Most `#include` statements are actually unnecessary. It is often adequate to forward-declare the class.

Wrong:

```
// myclass.h
// This pollutes include-hierarchy:
#include "otherclass.h"
class CMyClass: public CBase
    {
    ...
    private:
    COtherClass* iPointer;
    };
```

Including `otherclass.h` is unnecessary, and it pollutes the include hierarchy for `myclass.h`. A better solution would be:

```
// myclass.h
class COtherClass;
class CMyClass: public CBase
    {
    ...
    private:
    COtherClass* iPointer;
    };
```

Note that now only `myclass.cpp` has to include `otherclass.h`. Other modules using just `myclass.h` never need `otherclass.h`.

H File Template

```
/*
* Copyright (c) {Year(s)} {Company}.
* All rights reserved.
* This component and the accompanying materials are made available
* under the terms of the License "Symbian Foundation License v1.0"
* which accompanies this distribution, and is available
* at the URL "http://www.symbianfoundation.org/legal/sfl-v10.html".
*
* Initial Contributors:
* {Name} {Company} - Initial contribution
*
* Contributors:
* {Name} {Company} - {Description of contribution}
*
* Description:
* {Description of the file}
*
*/
```

```
// Protection against nested includes
#ifndef FILENAME_H
#define FILENAME_H

// System includes
#include <include_file>

// User includes
#include "include_file"

// Forward declarations
class forward_classname;

// External data types
/** description */
extern data_type;
//Global function prototypes

/**
 * description
 *
 * @param arg1 description
 * @return description
 */
type function_name( arg_list );
//Constants
/**  description /
const type constant_var = constant;
// Class declaration
/**
 *  one_line_short_description
 *  more_complete_description
 *
 *  @code
 *   good_class_usage_example(s)
 *  @endcode
 *
 */
class classname : public base_class_list
    {
    // Friend classes:
    friend class class1;
    friend class class2;
public:
    // Data types
    /** description /
    enum declaration
    /**  description /
    typedef declaration

    /** Constructors /
    IMPORT_C static classname* NewL();
    IMPORT_C static classname* NewLC();
    /**
     * Two-phased constructor.
     * @param arg1 description
     * @param arg2 description
     */
    IMPORT_C static classname* NewL( type1 arg1, type2 arg2 );

    /**> Destructor */
    virtual ~classname();

public:
    /**
     * description
     *
     * @param arg1 description
     * @param arg2 description
     * @return description
     */
    IMPORT_C type member_function( type1 arg1, type2 arg2 );

    /**
     * From base_class1.
     * description
     *
     * @param arg1 description
     */
    IMPORT_C type member_function( type arg1 );

protected:
private:
    classname();
    void ConstructL();

private: // Data
    /**
     * description_of_member
     */
    type member_name;

    /**
     * description_of_pointer_member
     */
    type* member_name;

    /**
     * description_of_pointer_member
     */
    type* member_name;
    };

// Inline functions
#include "include_file.inl"
```

HRH File

SFCSR-227: Place Declarations to HRH File

Usually only control IDs, defines and enums must be placed in the HRH file. Resource IDs will be collected automatically into `filename.rsg`, and must not be defined in the `.hrh` file.

HRH File Template

```
/*
* Copyright (c) {Year(s)} {Company}.
* All rights reserved.
* This component and the accompanying materials are made available
* under the terms of the License "Symbian Foundation License v1.0"
* which accompanies this distribution, and is available
* at the URL "http://www.symbianfoundation.org/legal/sfl-v10.html".
*
* Initial Contributors:
* {Name} {Company} - Initial contribution
*
* Contributors:
* {Name} {Company} - {Description of contribution}
*
* Description:
* {Description of the file}
*
*/
```

```
// Protection for nested includes
#ifndef FILENAME_H
#define FILENAME_H

// Data types
/** comment */
enum declaration

/** _comment */
typedef declaration

#endif
```

Implementation Files

CPP File

C/C++ source file.

SFCSR-230 Use CPP File Template

Use the CPP file template as a basis for the implementation file.

SFCSR-231 Implementation File Comments

Implementation file comments are written for the maintainer of the module. Comments are adequate if they:

Fulfill Coding Standards and Conventions/Programming Practices/Commenting.
Describe class-level implementation and clarify all the non-trivial issues.

SFCSR-232 Write the Implementation of One Class in One File

This makes maintenance and consistent commenting easier. Also, functions should be defined in the same order as in the header file.

SFCSR-233 Do Not Put Strings to be Localized into `.cpp` File

Use logical names in code files.
Put the actual strings in a separate resource header file (`.loc`),
that is, not to the same header file as resource structures (`.rh`).

CPP File Template

```
/*
* Copyright (c) {Year(s)} {Company}.
* All rights reserved.
* This component and the accompanying materials are made available
* under the terms of the License "Symbian Foundation License v1.0"
* which accompanies this distribution, and is available
* at the URL "http://www.symbianfoundation.org/legal/sfl-v10.html".
*
* Initial Contributors:
* {Name} {Company} - Initial contribution
*
* Contributors:
* {Name} {Company} - {Description of contribution}
*
* Description:
* {Description of the file}
*
*/
```

```
// System include files
#include <include_file>
// User include files go here:
#include "include_file"
// External function prototypes
extern external_function( arg_type, arg_type );
// Local constants go here:
const type constant_var = constant;
// ======== LOCAL FUNCTIONS ========
// --------------------------------------------------------------------------
// description
// --------------------------------------------------------------------------
//
type function_name( arg_type arg,
                    arg_type arg )
    {
    code  // implementation comment on this line
    // implementation comment on the following statement or block:
    code

// ======== MEMBER FUNCTIONS ========
//In the same order as defined in the header file.
// --------------------------------------------------------------------------
// description_if_needed
// --------------------------------------------------------------------------
//
classname::classname()
    {
    if ( condition )
        {
        // implementation_comment
        code
        }
    else
        {
        // implementation_comment
        code
        }
    // implementation_comment
    while ( condition )
        {
        code
        }
    // implementation_comment
    for ( for_init_statement; condition; expression )
        {
        code
        }
    // implementation_comment
    switch ( condition )
        {
        case constant:
            code
            break;
        case constant:
            code
            // fall-through intended here
        case constant:
            code
            break;
        default:
            code
            break;
        }
    }
// --------------------------------------------------------------------------
// description_if_needed
// --------------------------------------------------------------------------
//
void classname::ConstructL()
    {
    code
    }
// --------------------------------------------------------------------------
// description_if_needed
// --------------------------------------------------------------------------
//
EXPORT_C classname* classname::NewL()
    {
    classname* self = classname::NewLC();
    CleanupStack::Pop( self );
    return self;
    }
// --------------------------------------------------------------------------
// description_if_needed
// --------------------------------------------------------------------------
//
EXPORT_C classname* classname::NewLC()
    {
    classname* self = new ( ELeave ) classname;
    CleanupStack::PushL( self );
    self->ConstructL();
    return self;
    }
// --------------------------------------------------------------------------
// description_if_needed
// --------------------------------------------------------------------------
//
classname::~classname()
    {
    code
    }
// Non-derived function:
// --------------------------------------------------------------------------
// implementation_description
// --------------------------------------------------------------------------
//
```

INL File

C++ inline function definition file.

SFCSR-234 Inline Functions Must Be Defined in an Own File

Inline declarations should not be declared in the header file.

INL File Template

```
/*
* Copyright (c) {Year(s)} {Company}.
* All rights reserved.
* This component and the accompanying materials are made available
* under the terms of the License "Symbian Foundation License v1.0"
* which accompanies this distribution, and is available
* at the URL "http://www.symbianfoundation.org/legal/sfl-v10.html".
*
* Initial Contributors:
* {Name} {Company} - Initial contribution
*
* Contributors:
* {Name} {Company} - {Description of contribution}
*
* Description:
* {Description of the file}
*
*/
```

```
// Protection for nested includes
#ifndef FILENAME_H
#define FILENAME_H
// -----------------------------------------------------------------------------
// implementation_description
// -----------------------------------------------------------------------------
//
inline type classname::member_function()
    {
    code
    }
#endif
```

Resource Files

RH File

Resource header file.

SFCSR-240 Place Resource Structures in RH File

They do not follow C syntax, and thus cannot be compiled by the Ccompiler.
The related control IDs, defines and enums shall be placed in the HRH File.

RH File Template

```
/*
* Copyright (c) {Year(s)} {Company}.
* All rights reserved.
* This component and the accompanying materials are made available
* under the terms of the License "Symbian Foundation License v1.0"
* which accompanies this distribution, and is available
* at the URL "http://www.symbianfoundation.org/legal/sfl-v10.html".
*
* Initial Contributors:
* {Name} {Company} - Initial contribution
*
* Contributors:
* {Name} {Company} - {Description of contribution}
*
* Description:
* {Description of the file}
*
*/
```

```
// Protection for nested includes
#ifndef FILENAME_H
#define FILENAME_H
#endif
```

LOC File

The localization text file (LOC file) is a resource header file. It lists all text IDs (actually C macros) for texts to be localized.

SFCSR-241 Text ID

The syntax of a text ID is

```
// context_description_line(1)
// context_description_line(N)
//
#define text_<component>_freetext "text"
```

where

`context_description_line` is a description line clarifying the entry's context with information such as:
The context of the string: where exactly does/can the string appear?
Does the string appear, for example, in a dialog, menu, dialog button or notification?
What is done and by whom (user/application) for this text to appear?

Are there other texts/graphics in the same view/dialog/note?
Is a word a verb in the imperative, or is it a noun? For example, does the string `set` mean the verb 'to set' or the noun 'a set'?
What will replace `%U` (unicode text parameter) or `%N` (number parameter) included in texts? (For instance, is it a phone number or an e-mail address?)
`text_` starts a logical name. **Note:** The prefix `qtn_` has been used by the original contribution asset.
`component` is two to five lower-case characters derived from the component name.
`freetext` is the free text portion of the logical name. It may contain only lower-case letters ('a' to 'z'), numbers ('0' to '9'), and the underscore ('_'). The total length of the logical name must not exceed 50 characters.
`text` is the actual value.

Examples

```
// Command in options list in short-term memories.
// Opens the call list view that is focused.
//
#define text_logs_stm_cmd_open "Open"

// Title pane text in converter main state.
//
#define text_cnv_title "Converter"

// Prompt text for currency data query in converter.
//
#define text_cnv_edit_name_prompt "Currency name"

// Active call is terminated and a held call becomes active.
// Operation is confirmed with this info note.
// %U stands for the call identification of the activated call.
//
#define text_multc_unhold_done_note "%U active"
```

See also Coding Standards and Conventions/Programming Practices/Internationalization

RSS File

Resource file.

SFCSR-242 Write Resource Names in Lowercase

The resource compiler generates uppercase identifiers for C programs automatically.

SFCSR-243 Unique NAME
The NAME shall be unique among all the components in the build.

Use a source cross-reference tool to check that nobody is using the same name.

The maximum length is four characters.
SFCSR-244 Do Not Put Strings to be Localized into Resource Files
Use logical names in resource files, and
Put the actual strings in a separate resource header file (See this article, that is, not in the same header file as resource structures.
SFCSR-245 `TBUFnn` Variant
If you are using a `TBUFnn` variant, for example, `TBUF80` or `TBUF64`, and you are loading the string into a local `TBuf` variable, ensure that both 'nn's are equal. This prevents nasty buffer overflows at runtime. The preferred way is to use pure `TBUF` and use `StringLoader::Load` or `iEikonEnv->AllocReadResourceL` functions in your code.

RSS File Template

```
/*
* Copyright (c) {Year(s)} {Company}.
* All rights reserved.
* This component and the accompanying materials are made available
* under the terms of the License "Symbian Foundation License v1.0"
* which accompanies this distribution, and is available
* at the URL "http://www.symbianfoundation.org/legal/sfl-v10.html".
*
* Initial Contributors:
* {Name} {Company} - Initial contribution
*
* Contributors:
* {Name} {Company} - {Description of contribution}
*
* Description:
* {Description of the file}
*
*/
```

```
NAME four_letter_resource_name_in_uppercase_and_unique_in_component

// Component include files
#include "component.rh"
#include "component.hrh"
#include <component.loc>
#include <component.mbg>

// Common include files
#include <eikon.rh>
#include <avkon.rh>
#include <avkon.loc>
#include <avkon.rsg>
#include <data_caging_paths_strings.hrh>

// Resource identifiers
RESOURCE RSS_SIGNATURE { }
RESOURCE TBUF { buf=""; }

// Constants
#define constant value

// ----------------------------------------------------------------------------
// resource_name
// description
// ----------------------------------------------------------------------------
//
RESOURCE structure_name [resource_name]
    {
    resource_data; // comment

    // comment
    resource_data;
    };
```

Registration RSS File Template

The application registration file defines information about an application that is required by the application launcher or system shell.

```
/*
* Copyright (c) {Year(s)} {Company}.
* All rights reserved.
* This component and the accompanying materials are made available
* under the terms of the License "Symbian Foundation License v1.0"
* which accompanies this distribution, and is available
* at the URL "http://www.symbianfoundation.org/legal/sfl-v10.html".
*
* Initial Contributors:
* {Name} {Company} - Initial contribution
*
* Contributors:
* {Name} {Company} - {Description of contribution}
*
* Description:
* {Description of the file}
*
*/
```

```
// Use standard includes
#include <appinfo.rh>
#include <component.rsg>
#include <data_caging_paths_strings.hrh>

UID2 KUidAppRegistrationResourceFile
UID3 0x // Application UID

RESOURCE APP_REGISTRATION_INFO
    {
    app_file = "component";
    localisable_resource_file = APP_RESOURCE_DIR/component";
    localisable_resource_id = R_component_LOCALISABLE_APP_INFO;
    }
```

Extension Files

Extension makefiles can be used where certain build steps are required that are not catered for by the generated makefiles. The makefiles must contain certain make targets. During build activities, `abld` will call the target corresponding to the build activity that is being carried out. This will then execute whatever commands the makefile specifies for that target. Extension makefiles must not contain or call any scripting or programming languages other than those supported by the native Symbian tool chain. Acceptable programming languages or environments are:

Dos batch files (`.bat` or `.cmd`)
Perl scripts (`.pl` or `.pm`)
Makefiles (`.make` or `.mk`)

For details about build system, please refer Developer Guidelines: Foundation Builds

Programming Practices

Writing Good Code

Consistency

Be consistent in the use of programming conventions (or coding guidelines).

When modifying a big chunk of third-party code, you should adhere to the coding conventions observed in the third-party code, rather than apply host conventions – like those detailed here.
Include the header file for the source as the first `#include` in the source file.
Include only the needed header files.

SFCSR-300 Use Source File Templates

The templates include sample content for each source file type. The templates are there to

Help creating real source files in a correct manner
Help obeying key recommendations
Help in proper documentation.

Use the page about filetypes to help in navigating various source file types and corresponding templates.

SFCSR-301 Include Statement

In the `#include` statement:

All slashes in include directives must be forward slashes (/).
All references to directory paths and to filenames must be in lowercase.
All references to directories shall be
relative to `/epoc32/include`, or
relative to **common include directory** which is available via the layer-specific SYSTEMINCLUDE path.

Examples:

```
// Include Publish & Subscribe header (relative to /epoc32/include)
#include <e32property.h>

// Include ECOM header  (relative to /epoc32/include)
#include <ecom/ecom.h>

// Include help ids (relative to common include directory)
#include <csxhelp/sms.hlp.hrh>
```

SFCSR-302 No Location Assumptions

Package shall not assume anything from its location in source tree. All compile-time dependencies should be to `/epoc32/include` or inside the package where relative paths shall be used.

The package should also compile for instance in the `/somedir/somedir2/somedir3/<package>` directory **without** the /sf-directory even existing in build area.
package collection **component** must not use static path definitions, since it may be used in an arbitrary location in the foundation source tree.

SFCSR-303 Use Source-Analysis Tools

Use available code-analysis tools to detect and fix basic errors before contribution.

SFCSR-304 No Compilation Warnings

Code must compile with no warnings.

Build code that gives zero compiler warnings and zero link errors; it is lazy to ignore compiler warnings, and these are highly likely to generate errors on other compilers.
Complacency regarding compiler warnings leads to important new compiler warnings not being noticed among a pile of familiar warnings that are 'taken for granted'.

Meaningful names

Use meaningful names for classes, methods and data members.
Carefully chosen names can make most code self-documenting.

Example:

Here are four versions of the same line of code:

```
iFlag( ETrue ) ; // Set iFlag
iFlag( ETrue ) ; // incoming datatype is unknown
iFlag( ETrue ); // incoming data type has been recognised
iDataTypeIsRecognised( ETrue );
```

The first line is a waste of space and tells us nothing.
The second line is worse than nothing at all, because the comment is wrong and there is no easy way to tell.
The third line is acceptable because the comment is correct.
The last line is best because it explains what is going on in every use of iDataTypeIsRecognised. This code comments itself.

No Magic Constants

See here.

Punctuation

Use spaces around/after operators (including brackets) to improve the legibility of code.

Correct:

```
for ( TInt ix = 0; ix < nbrOfProperties; ix++ )

void CSomething::NewL( TInt aParam, TTime aTime, TInt aRepeatCount )

x += 5 * ( y / ( 27 - y ) );

TRAP( err, aBytesThroughFile += ( size - stream.Source()->SizeL() ) );

if ( ( iCurrentChar == ECarriageReturn ) && ( lastChar != KVersitTokenBackslashVal ) )
```

Incorrect:

```
for(TInt ix=0;ix < nbrOfProperties; ix++)

void CSomething::NewL(TIntaParam,TTime aTime,TIntaRepeatCount);

x+=5*(y/(27-y));

TRAP(err,aBytesThroughFile+=(size-stream.Source()->SizeL()));

if((iCurrentChar==ECarriageReturn)&&(lastChar!=KVersitTokenBackslashVal))
```

Efficiency

Write code that is lean and efficient. For example,

Minimise stack use as much as possible.
Avoid continuous memory allocating and deleting
Consider resetting and re-using objects instead of deleting.
Consider using of own memory pool especially if lots of dynamic memory allocation of the same-sized objects is needed.
Utilize Symbian OS Hash table classes such as `RHashSet, RPtrHashSet, RHashMap` or `RPtrHashMap`
Avoid calculations within the loop.
If calculation is not influenced by anything that happens within the loop, simply move it outside the loop, and keep re-using that value inside the loop.
Read and Write larger data blocks at one go. The preferred block size is multiple of 512 bytes, 512, 1024, 1536 etc. bytes. Not, for example, 5 or 10 bytes.
Do not use `TFindFile::FindWildByDir` and `TFindFile::FindByDir` by accident. It searches for a file in a directory on all available drives.
Be aware of the size of TFileName and do not needlessly make copies.
Consider batching multiple changes so that they may be optimized, for example, by using `DrawDeferred` rather than `DrawNow` when updating a window.

See more from Guide to Optimising Performance (http://developer.symbian.org/sfdl/doc_source/guide/Performance/GuideToOptimisation.html)

Compatibility

Please see Preserving Compatibility. These guidelines explain how to use *in-source* documentation conventions to document interfaces within source code. The goal is to ensure that interfaces are accurately and well documented, in a way that supports automated documentation builds as part of the productization process. **Documenting APIs is essential to their usability and reusability.**

These guidelines are aimed at software engineers and authors. C++ is assumed to be the implementation language.

General Commenting Guidelines

Remember that, in most cases, the person reading your comments is a developer who is trying to understand the interface, or implementation, you have provided. Therefore:

Write the kind of documentation you would like to read.
Clarity is key.
Remember that if an interface isn't documented adequately, it can't be used properly.
Even where you think the behavior of a function or the purpose of a class is obvious from its name, you should still document it.

You can make your writing clearer by observing these rules of thumb:

Write in standard US English.
Use the present tense.
Write in the third person, for example: 'The function panics if 0 is passed in.'
In-source documentation comments in header files should explain how to use each class or other declaration. Using the Doxygen tags @code and @endcode is a good way to add usage example blocks into the comments.
Comments in implementation files should explain *why* something has been done, rather than saying *what* has been done. Non-obvious control and data structures should be explained, as well as the programmer's assumptions of the program's state at a certain point, if these are not obvious.
Do not mention company-internal details that users of the interface will not have access to, for example:
Internal extension class names
Internal tools' IDs
Internal project codes.
If you are referring to existing OS or industry concepts or terms, check that you are using the standard form of names for these terms.
If you are documenting a new interface that is introducing new concepts into the OS, ensure that you use the names for such concepts consistently. Use the same definitions as in the related design and engineering product documentation.
Acronyms and names should be correctly capitalized (for example, 'TCP/IP' or 'Unicode', rather than 'tcp/ip' or 'unicode').
If you need to refer to a layer, package, collection or component, refer to it by its name in the source tree.
Avoid inventing names to refer to groups of items (for example, the 'foo API') that are not defined in the model.
Use 'this' or 'the object' to refer to the parameter passed as the 'this' pointer.
Keep comments up to date.

Since essential parts of the API documentation will be generated from in-source comments, it is crucial to review the documentation properly both technically and from a language point of view.

SFCSR-310 Standard File Header

Every SFL licensed source file created manually must contain the standard file header based on the file header template below.

```
/*
* Copyright (c) {Year(s)} {Company}.
* All rights reserved.
* This component and the accompanying materials are made available
* under the terms of the License "Symbian Foundation License v1.0"
* which accompanies this distribution, and is available
* at the URL "http://www.symbianfoundation.org/legal/sfl-v10.html".
*
* Initial Contributors:
* {Name} {Company} - Initial contribution
*
* Contributors:
* {Name} {Company} - {Description of contribution}
*
* Description:
* {Description of the file}
*
*/
```

(The comment character varies per file type.)

All the the {tag}s must be filled in with correct values.

You can find further explanation of copyright and tags here.

Comment Block Formats

When writing in-source documentation, comments in C/C++ code use syntax like this:

```
/**
Doxygen documentation comment style example.
The second line of comment.
*/
```

or this:

```
/**
* Doxygen comment style one line example.
* The second line of comment.
*/
```

Note
Only in-source documentation within Doxygen delimiters, `**` and `*/`, and adhering to the guidelines below, will appear in the Developer Library.

When writing normal source code comments in C/C++ code, use syntax like this:

```
// C++ comment style example.
// The second line of comment.
```

or this

```
/*
C comment style example.
The second line of comment.
*/
```

```
/*
* C comment style example
* The second line of comment.
*/
```

In-Source Documentation

In-source documentation consists of **structured, marked-up comment blocks** within the source code. It is built by documentation generation tool(s) to produce production-quality API-reference documentation for publication in Symbian Foundation releases.

Its goal is to ensure that interfaces are accurately and well documented, in a way that:

Supports automated documentation builds as part of the productization process
Helps contributors to use and contribute to Symbian Foundation software assets
Helps package owners to understand contributions
Supports automated generated documentation builds as part of the productization process
Supports the Doxygen documentation tool.

In-source documentation also helps you as follows:

By documenting your code, you can stop other people using it badly – you can help prevent other people's defects before they are written.
Documenting as you code is like peer review. It can expose your own assumptions, clarify your design choices, highlight code complexity, encourage openness – it can help stop your own defects before they are written.
Enables a closer coupling of code and documentation, allowing concurrent updates of both.
Note
The delimiters of Doxygen comment blocks, `/**` and `*/`, differ to those of standard C/C++ comment blocks `/*` and `*/`. See here for further information. Only in-source documentation within Doxygen delimiters, and adhering to the guidelines below, will appear in the Developer Library.

What to Document

As a general recommendation, in newly written code, in-source documentation **must** be provided for

every class that `EXPORT`s a function
every class that includes a pure-virtual function that is intended for derivation
friend classes that don't `EXPORT` but that manipulate documented classes
all public `EXPORT`ed members of documented classes
every `EXPORT`ed function that is controlled by security capabilities
other non-member items that are intended to be part of an interface.

Usually, private members do not require documentation because in-source documentation is intended to document interfaces, not implementations.

Where to Document

As a general recommendation document

classes at class level in the header file at the point of declaration
class data members in the header file at the point of declaration
pure virtual functions in the header file at the point of declaration
class member functions either in the header file or in source files in which they are implemented. Follow the way a package uses, in general.
other non-member items (global data, global functions, enums, typedefs, macros) in the header file at the point of declaration.
Note
You should not document items at both their point of declaration and of definition or split your documentation between the two. Doxygen will get the documentation from one location only, and will not merge the two.

The comment for an item should be inserted before the item being commented (only white space and standard C++ comments are allowed in between). Note that, at one time, comments were positioned just before a function or class opening brace: the tools will still extract such comments, but any new comments should be added before the item.

Comment Block Format

In-source documentation is written in a comment block introduced and closed by the special comment pair delimiters: **/**** and ***/**.

Example:

```
/**
In-source comment about the class CPpopertyObserver
*/
class CPpopertyObserver: public CGenericObserver
    {
public:
    // Other declarations
    }
```

Usually, it's most readable if delimiters are placed on new lines, on their own, but this is not required.

Within the delimiters, the documentation block consists of:

Plain text overview: one or more lines, terminated by a tagged line, or the end of the comment. It describes the purpose, usage and behavior of the item. Overview text should include example code fragments where this helps to clarify the use or behavior of the interface. Overview text can have multiple paragraphs: use a blank line (that is, two new-line characters) to separate paragraphs.

Overview text should always include example code fragments where this helps to clarify the use or behavior of the interface. Example code fragments should be contained within @code and @endcode tags.

Tag sections: one or more lines starting with an `@` tag from the set described in the supported tag set, followed by arguments where necessary, terminated by a blank line, or another tagged line, or the end of the comment. Each tag section describes a particular aspect of the interface provided by the item.

Tag sections can occur in any order, and can be placed before or after the overview. Do not interleave tag sections between overview paragraphs: keep the overview together.

If one or more tags precede the overview, leave a blank line before the overview (required for Doxygen compatibility).

Avoid complicated formatting (don't start lines with an asterisk, don't box text with asterisks); it makes things harder for future maintainers.

Aligning all documentation at the left will make it more readable, but this is not required.

Note
C-style box comments like `********** ... ********/` will **not** be picked up by Doxygen. Therefore, to avoid any possible confusion, we recommend that you do not use this style of commenting. Where legacy or bought-in code uses this comment style, there is no requirement to remove it.

Documenting Classes

Comment location

Document a class at class level in the header file in which it is declared.
The documentation block should precede the class declaration.
Align the block with the C++ 'class' keyword, to save maintainers from having to tab or space comment lines in later amendments.

Comment contents

Overview text must be provided describing the class.

Purpose: a single sentence summarizing the purpose of the class. This should allow the user to decide quickly if the class provides the type of functionality that they require or not. Some guidelines for this:
If a class encapsulates some data and operations on that data, then provide a short description of the data, for example: `A database of contact items`.
If a class primarily provides a collection of utility functions, rather than encapsulating a data type, use a verb to describe what the utilities do, for example: `Sorts contacts by specified criteria`.
If a class primarily defines an interface (an M class), then this can be stated explicitly, for example: `Defines the interface for notification of events on a contacts item`.
Detail what the class represents. For example, if the class represents a contact item, then describe the properties that a contact item has. If the class implements a standard, or part of a standard, this should be described. As part of this, describe any key relationships with other classes that are useful for the user to understand the class. For example, if the class defines a callback interface, then say which classes (if known) are expected to call the interface. If the class manages a collection of objects, say what object type is being managed.
If a class has an unusual lifecycle, for example, a C class that is not created through `NewL()`/`NewLC()` functions, then describe this.
Do not describe internal design that is not visible to the user of the class.
For larger classes, for example, with more than five members, it can be very useful to provide a breakdown of its items into related groups of functions. For example:

```
/**
...
OpenL(), Close(), CompactL(), DeleteL() are used to manage a database file.

NewItemL(), DeleteItemL(), and EditItemL() are used to change the items in the database.

LastError() and Size() get properties of the database.
...
*/
```

If a class provides a mixture of library functionality (functions that are called) and framework functionality (functions intended to be re-implemented), the class description should clearly describe both uses.

To improve clarity, it is better to use the following tags if applicable:

@code and @endcode – a block of code illustrating uses of the class. This is recommended as sometimes being more concise and easily understandable than explaining the same thing in descriptive text.
@see – cross-references to related interface items.
Note
Since the first sentence of the comment will be auto-extracted as a brief description of the class, the first sentence should be as descriptive as possible. See the example below.

Example

```
/**
Provides information on which features are supported in the environment.

A feature is an area of functionality that may or may not be present on a phone,
depending on the configuration the device vendor has made.
Each feature is identified by a UID defined in features.h.

If querying only one feature, it is more efficient to use the class via the static function,
IsFeatureSupportedL(). When querying more than one feature, it is more efficient to use the
class by creating an instance and calling the IsSupported() function.

@code
// Static way of using the class:
TBool isSupported = CFeatureFinder::IsFeatureSupportedL( KFeatureIdUsb );

// Dynamic way of using the class using NewL():
CFeatureFinder* testA = CFeatureFinder::NewL();
TBool usbSupported = testA->IsSupported( KFeatureIdUsb );
TBool mmcSupported = testA->IsSupported( KFeatureIdMmc );
@endcode

*/
class CFeatureFinder : public CBase
    { ...body of class declaration    };
```

```
/**
Plain text, the overview for this function.

A detailed description of the function.

@param aParam1 Explanation of this parameter
@param aParam2 Explanation of this parameter
@return        Explanation of the object returned
@pre  None
@post CClassExample object is now fully initialized
*/
EXPORT_C TType CClassExample::Foo( TBool aParam1, COther& aParam2 )
    { ...body of function implementation goes here    }
```

Example 2

The following is an example of a fully documented function.

Note: Normally, only a few tags would be required, but this example illustrates the use of most of the recommended tags in one single example.

```
/**
* Obtains the identifier of the system's default positioning module.
* This module will be used if no criteria is specified when an
* RPositioner sub-session is created.
*
* @param aModuleId    This will be set, upon successful completion, to the ID
*                   of the module that the positioning server uses by default.
* @return           A Symbian OS error code.
*
* @panic EPositionServerHandleNotClosed An attempt has been made to connect
*                                       using an already open handle.
* @leave KErrPositionBufferOverflow     There is insufficient space to return the
*                                       required information back to the client.
* @leave KerrConnectionFailure          Failed to connect to location server
*
* One small sample describing the use of the function.
* @code
* CRequestor* service = CRequestor::NewLC(
*    CRequestor::ERequestorService,
*    CRequestor::EFormatApplication,
*    KSrvName );
* stack.Append( service );
* @endcode
*/
IMPORT_C TInt GetDefaultModuleId( TPositionModuleId& aModuleId ) const;
...
```

Note

If an implementation of a pure virtual function is undocumented, then the documentation tools will automatically provide it with any documentation provided for the pure virtual function. You can use this feature to save copy/pasting comments, but in most cases the implementation will have distinctive behavior such as panics and errors that the user will want to know about, and you should document the implementation function separately.

Documenting Pure Virtual Functions

Comment location

Pure virtual functions are an exception to the general recommendation of documenting functions with their implementation, because they have no immediate implementation. Therefore, document them in the header file in which they are declared.

Comment contents

See the the documenting conventions for guidelines on what to document.

Example

```
/**
Creates a bitmap for the specified Unicode character.

Implementations of this function should put the bitmap in
aGlyphData->iBitmapBuffer, and the character metrics are placed in
aGlyphData->iMetricsBuffer. The other parts of aGlyphData should be
left alone.
...etc., more documentation follows
*/
virtual void RasterizeL( TInt aCode,TOpenFontGlyphData* aGlyphData ) = 0;
```

Documenting Member Data

Comment location

Document member data in the class declaration in the header file.

In all cases, place the documentation before the item. This provides a narrative introduction to someone reading the source code.

Comment contents

Document each public data item. Usually, only a brief overview is needed and the whole documentation block can be kept to a single line:

By default, private member data is assumed not to be part of the interface and need not be documented in-source, though it is good practice to comment it.

Example

```
public:
    /** Brief overview, what this TInt represents */
    TInt iMember;

    /** Internal state data. */
    TInt iMember2;
```

Example

Note that the example members are documented, even though they are in the scope of the `private` keyword.

```
/**
....etc., more class level documentation here
*/
class REComSession : public RSessionBase
    {
    ....etc., more class declaration information here

private:
    /**
    * Flag indicating that this session should not be closed because
    * the interface implementation is in the middle of construction and
    * this will be on the CleanupStack
    */
    TBool iDoNotCloseThis;

    /** Buffer for the arguments to the framework services */
    TAny* iServiceArgs[KMaxServiceArgs];

    /** Handles the loading of the DLLs and the deletion of objects */
    CLoadManager* iLoadManager;
    };
```

Notes

There is no support for applying a single comment to a group of members.
If you decide that for some reason private member data requires documentation, follow the same recommendations as for public member data. (Examples might be where it is expected that external parties will re-implement functionality, and not just use interfaces.)

Documenting Enums

Comment location

Document enums in the header file in which they are declared.

In all cases, place the documentation before the item. This provides a narrative introduction to someone reading the source code.

Comment contents

Provide overview text for the enumeration itself, and for each enum member.

Use @see to provide a link to related items: for example, if the enum values are declared for use with a particular function, provide a link to that function.

Example

```
/**
Class-level documentation here
*/
class TOpenFontSpec: public TOpenFontFaceAttribBase
    {
public:
    ...etc., more class-declaration information here
    /**
    Algorithmic effects flags.

    These can be combined using an OR operation.
    */
    enum TAlgorithmFlags
        {
        /** Font is algorithmic bold. */
        EAlgorithmicBold = 1,
        /** Font has a drop shadow. */
        EDropShadow = 2,
        /** Font is an outline font. */
        EOutline = 4
        };
    ...etc., more class-declaration information here
    };
```

Example

Documenting after the declaration using `//` comments. This is preferred if comments are short. This increases the readability of the header file.

Note
The length of the line must not exceed 78 characters. If this is likely to happen use `/**...*/` instead.

```
/**
* Battery status of device.
*/
enum EPSHWRMBatteryStatus
    {
    EBatteryStatusUnknown        = -1, // Uninitialized or some other error
    EBatteryStatusOk             = 0,  // Used during charging
    EBatteryStatusLow            = 1,  // Show note to user Battery low
    EBatteryStatusEmpty          = 2   // Show "recharge battery" note to
    };
```

Example Use `/**...*/<` comments for longer descriptions of enumeration values (when the line length exceeds 78 characters).

```
/**
* Defines the types of file compression.
*/
enum TBitmapfileCompression
    {
    /** Bitmap file is not compressed. */
    ENoBitmapCompression = 0,
    /** File is compressed using run-length encoding compression. */
    EByteRLECompression = 1,
    /** File is compressed using 12-bit run-length encoding compression. */
    ETwelveBitRLECompression = 2,
    /** File is compressed using 16-bit run-length encoding compression. */
    ESixteenBitRLECompression = 3,
    /** File is compressed using 24-bit run-length encoding compression. */
    ETwentyFourBitRLECompression = 4
    };
```

Documenting Global Items

Comment location

Document global data, typedefs and macros in the header file in which they are declared.

In all cases, place the documentation before the item.

Comment contents

Provide overview text for each item.

Use @see to provide a link to related items: for example, if a constant is used with a particular function, provide a link to that function.

Example

```
/**
Defines the priority associated with a data type.
The priority is used by a device to resolve the current preferred
handler of a data type, in the absence of any user preferences.

@see TDataTypeWithPriority
*/
typedef TInt16 TDataTypePriority;
```

Example

Constants are documented in the header file, for example:

```
/**
* Maximum allowed intensity setting for light
*/
const TInt KHWRMLightMaxIntensity(100);
```

```
/**
* Bluetooth power setting API
*
* Possible integer values:
* - 0 (EBTSetPowerOff) Set BT power to ON
* - 1 (EBTSetPowerOn)  Set BT power to OFF
*/
const TUint KBTSetPowerState = 0x00000001;
```

Documenting Namespaces

Comment location

Document a namespace in one location, in an `EXPORT`ed header file.

Namespaces are unusual in that their contents can be set in multiple header and `.cpp` source files. If this is the case, the other owners of all the locations should agree in which file the in-source documentation should be located.

Comment contents

Overview text should describe what the namespace includes.

Example

```
/**
WAP-related types and error codes.
*/
namespace Wap
    {
    /**
    documentation for WapParser
    */
    class WapParser: public CBase
    {
    }
```

Doxygen Tags

This section lists the Doxygen tags to be used for API in-source documentation.

Note
The list below does not mention other tags present in the initial contribution asset.

SFCSR-311 Essential Doxygen Tags

No tags are strictly mandatory, as the code will still compile even with no tags present. We cannot enforce documentation and tagging of APIs, but these tags are considered **essential** for usability and reusability.

Platform security tags:
@capability and @ref

SFCSR-312 Strongly Recommended Doxygen Tags

It is strongly recommended that the following tags are used in in-source comments:

@code and @endcode
@param
@return
@leave
@panic
@pre
@post
@see

Text Formatting and Special Characters

You can use various tags to format descriptive text, and insert special characters:

Lists

Doxygen has a number of ways to create lists of items; the following are recommended:

Use of the HTML format is recommended for list items that consist of multiple paragraphs.
Use of Doxygen dashes is recommended for list items consisting of only one paragraph.

The following tags can be used to create lists:

Tag	Description
`@li`	Creates an item bullet in a list. This command has one argument that continues until the first blank line or until another `@li` is encountered. The command can be used to generate a simple, not nested list of arguments.
`-`	Similar to `@li` above, but can be used to created nested lists with the use of indentation.
`-#`	Treated the same as `-` above, but will give a numbered list (or lettered or roman numeralled list depending on the nesting).

Example with dashes

```
/**
A list of events:
- mouse events
    -# mouse move event
    -# mouse click event
- keyboard events
    -# key down event
    -# key up event

More text here.
*/
```

The text above creates this output:

A list of events:
mouse events
mouse move event
mouse click event
keyboard events
key down event
key up event

More text here.

The following HTML tags can be used for lists:

Tag	Description
<LI>	Starts a new list item.
</LI>	Ends a list item.
<OL>	Starts a numbered item list.
</OL>	Ends a numbered item list.
<UL>	Starts an unnumbered item list.
</UL>	Ends an unnumbered item list.

Example with HTML

A list of events:

```
  <ul>
  <li> mouse events
     <ol>
     <li>mouse move event
     <li>mouse click event\n
         More info about the click event.
     <li>mouse double click event
     </ol>
  <li> keyboard events
     <ol>
     <li>key down event
     <li>key up event
     </ol>
  </ul>
  More text here.
```

Linking to Other API Items

Within the same component

If you mention the name of another API item in descriptive text, and this item exists in the same component, then the name will become a hyperlink in the output.

C++ scope is understood by the tools, so if you want to refer to another member in the same class, just use the member's name.
Use empty brackets when referring to function names, for example: `TDes::Append()`. Specify the parameter types for overloaded functions to avoid ambiguity.

Links are automatically made to the types used in function parameters and returns.

To see an example of linking to an another function within a function comment see the first @pre example below.

In other components

You can create a hyperlink to any other API item in the Developer Library using the `@see` tag. See the @see section for more information.

Linking to Other Documentation

Links to websites can be made simply by giving the URL without any markup. For example:

```
/**
Sets the type attribute of the object.

This should be in the form of a valid IANA media type (see http://www.iana.org/assignments/media-types/index.html).
*/
IMPORT_C void SetTypeL(const TDesC8 &aDesc);
```

Code Font Character Formatting

Tag	Description
`@c` *word*	Used to make the *word* following the tag appear in code font. The space after the *word* finishes the effect.
`@p` *word*	Treated the same as `@c` above.

Other Formatting Tags

The following table contains other useful formatting tags:

Tag	Description
<B>	Starts a piece of text displayed in a bold font.
</B>	Ends a bold font section.
<CENTER>	Starts a section of centered text.
</CENTER>	Ends a section of centered text.
<I>	Starts a piece of text displayed in an italic font.
</I>	Ends an italic font section.

Escaped Characters

Certain characters have special meanings to Doxygen and need to be escaped (using a backslash) in order to achieve the desired output. These are listed in the following table.

Tag	Description
\\	Writes a backslash character.
\@	Writes an at-sign (@).
\&	Writes the & character to output.
\$	Writes the $ character to the output.
\#	Writes the # character to the output.
\<	Writes the < character to the output.
\>	Writes the > character to the output.

Function Interface

The following tags document the signature of a function, explaining its parameter and return values.

@param

Purpose

Describes a function parameter.

Syntax

Tag	Description
`@param` *name description*	Parameter *name* is used in this function as described in *description*. *name* is the identifier only, not the type; for example in `TInt aArg<`, the *name* is `aArg`. If the parameter is unnamed then use `#` for the *name*, then the particular `@param` command will be treated by *ordinal* (see below). *description* is **required** and consists of plain text line(s) terminated by either a blank line, or the next `@` tag in the comment block, or the end of comment block delimiter.

Usage

The function description must be provided even where it seems obvious.

The existence of the parameter is checked and the Doxygen warning log shows if the documentation of this (or any other) parameter is missing or not present in the function declaration or definition.

For all non-const parameters, state how the function might alter them. If a parameter is a reference and returns with a different value, then write `On return, .....`

Try to give the range of possible values for the parameters.

TRequestStatus is a very common parameter type. The following form of words is suggested for describing it:

```
Asynchronous request status. On request completion, KErrNone if successful; otherwise, a system-wide error code.
```

You should modify this to describe any particular error codes that are relevant to the function.

Only **one** instance of this tag should be used per parameter. Where the parameter is of a type which has, for example, multiple fields, then either:

Describe the generic value – for example a `TPositionParam` could be documented as `...text/glyph data for the algorithm....` A link will be auto-generated from the type to the reference documentation for the type which describes the field values in detail.
Put the necessary extra detail in the descriptive text for the function.

For nameless parameters use `#` for the `name`. In that case, the parameter refered to has the same ordinal value as the position of the `@param` command. These can be mixed with named `@param` commands.

Example

```
/**
Doubles a specified integer.
@param aInput   Value to double.
@param aDoubled On return, the value of aInput * 2.
*/
void Foo::Double( TInt aInput, TInt& aDoubled )
```

Example of `@param` by ordinal

```
/**
@param # Stuff about the first parameter
@param # Second parameter stuff.
@param # Documentation about the last one.
*/
IMPORT_C int socket( int, int, int );
```

The following example is fine because the parameter-by-ordinal does not conflict with parameter-by-name. (Note that the names are reversed – this is discouraged but will be resolved by the tools!)

```
/**
@param b Stuff about parameter b
@param # Second parameter stuff.
@param a Documentation about parameter a
*/
IMPORT_C int socket( int a, int, int b );
```

The `@param` command has an optional attribute specifying the direction of the attribute. Possible values are *in* and *out*. Here is an example for the function `memcpy()`:

```
/**
Copies bytes from a source memory area to a destination memory area,
where both areas may not overlap.
@param[out] dest The memory area to copy to.
@param[in]  src  The memory area to copy from.
@param[in]  n    The number of bytes to copy
*/
void memcpy( void *dest, const void *src, size_t n );
```

If a parameter is both input and output, use `[in,out]` as an attribute.

@return

Purpose

Describes a function's return value.

Syntax

Tag	Description
`@return` *`description`*	Value is returned as described in *`description`*. **No type information is given.** `@return` is required unless the function returns void *`description`* is required and consists of plain text line(s) terminated by either a blank line, or the next `@` tag in the comment block, or the end of comment block delimiter.

Usage

A function should have at most one `@return`. If you want to describe multiple possible return values, do this in a single sentence, for example:

```
KErrOutOfRange if an out of range array index is specified; KErrNotFound if the array has not been constructed
```

If multiple adjacent `@return` commands are given they will be joined into a single paragraph.

For the return value description, don't write '`Returns the...`' just write what it is.

Suggested standard words for a `TInt` error code return value are

```
KErrNone if successful, otherwise one of the system-wide error codes
```

Example

```
/**
Gets the size of the array.
@return The size of the array, or KErrNotFound if the array has not been initialized.
*/
TInt Foo::Size()
```

Function Behavior

@pre

Purpose

Describes a pre-condition for using a function.

Syntax

Tag	Description
`@pre` *`explanation`*	*`explanation`* describes what relevant conditions must hold on entry to this function, including what assumptions have been made and what, if anything, should have been done before calling this function. *`explanation`* always consists of plain text line(s) terminated by either a blank line, or the next `@` tag in the comment block, or the end of comment block delimiter.

Usage

Pre-conditions help in providing clear and unambiguous descriptions of semantics. Documenting pre-conditions is also a useful tool to help you to refine the design – having many pre-conditions may imply over-dependence on state.

If the function semantics are already described fully by @param and @return tags, do not use this tag.

Pre-conditions fall into two categories:

Sequential preconditions
Pre-conditions linked to a state

Sequential pre-conditions

Sequential pre-conditions state the events that need to have happened before the API can be called. In the simplest form, the pre-condition lists the APIs that need to have been called previously.

Example

```
/**
Completes and returns an encoded field 8-bit buffer.

The final buffer containing the entire encoded header is constructed.
The function returns a buffer containing the encoded field constructed
from the first call to StartHeaderL().

@pre    The function StartHeaderL() should have been called.
@post   Encoder is reset ready to be used again.
@return Pointer to a buffer containing the entire encoded field.
        Responsibility for de-allocating the memory is also passed.
*/
EXPORT_C HBufC8* CWspHeaderEncoder::EndHeaderL()
```

Pre-conditions linked to a state

Pre-conditions linked to a state are those pre-conditions that are linked to a state mandate, when the system has discrete states, and when the API can only be called while the system is in a given state.

Example

```
/**
Creates a nanothread.

This function is intended to be used by the EPOC kernel and by personality
layers. A nanothread may not use most of the functions available to normal
Symbian OS threads. Use Kern::ThreadCreate() to create a Symbian OS thread.

@pre The call is in a thread context.
@pre Interrupts must be enabled.
@pre The kernel must be unlocked.

@param aThread Description of the parameter
@param aInfo   Description of the parameter
@return        Description of what's returned
*/
EXPORT_C TInt NKern::ThreadCreate( NThread* aThread, SNThreadCreateInfo& aInfo )
```

@post

Purpose

Describes a post-condition of a function.

Syntax

Tag	Description
`@post` *`explanation`*	*`explanation`* describes what relevant conditions will hold on exit from this function, where these are not already covered by the `@return` and `@param` tags. *`explanation`* always consists of plain text line(s) terminated by either a blank line, or the next `@` tag in the comment block, or the end of comment block delimiter.

Usage

Post-conditions help in providing clear and unambiguous descriptions of semantics.

If the function semantics are already described fully by @param and @return tags, do not use this tag.

Example

See the first @pre example above.

@leave

Purpose

Describes a leave condition of a function.

Syntax

Tag	Description
reason description</code>	Function will leave with `reason`*, which is either a leave code or the name of the L-function called, in circumstances explained by* `description`*.*

	description is required where relevant to a required capability, otherwise optional. *description* always consists of plain text line(s) terminated by either a blank line, or the next @ tag in the comment block, or the end of comment block delimiter.

Usage

A separate `@leave` tag should be used for each leave code that the function may generate.

Normally, any leave where the reason is defined within this interface should be documented. Some exceptions are

Out-of-memory (OOM) conditions may be assumed to cause a leave without further documentation, but this might depend on context.
Leaving with a standard error like `KErrNotFound` may not require further documentation, depending on context.

Example

```
@leave KErrNotFound aMtmTypeUid does not specify a registered MTM.
```

Example

```
/**
Two-phased constructor for implementation class.
Use this function for creating a Light client with callbacks.
Leaves instance to cleanup stack.
@param aCallback        Pointer to callback instance
@return                 New CHWRMLight implementing instance.
@leave KErrNotSupported Device doesn't support Light feature.
@leave KErrNoMemory     Memory allocation failure.
*/
IMPORT_C static CHWRMLight* NewLC( MHWRMLightObserver* aCallback );
```

@exception

Purpose

Describes an exception condition of a function.

Syntax

Tag	Description
`@exception` *name description*	Function will raise an exception with the *name* given in circumstances explained by *description*. *name* and *description* are required. *description* always consists of plain text line(s) terminated by either a blank line, or the next @ tag in the comment block, or the end of comment block delimiter.

Usage

A separate `@exception` tag should be used for each exception that the function may generate.

Any ANSI C++ function can raise the exception.

Example

```
/**
Creates an SVG document from the given string.
(plus more documentation ...)

@param aContent         Valid SVG formatted string from which the SVG content should be read.
@exception SvgException Creating the document failed.
*/
__declspec(dllimport) SvgDocumentHandle createSvgDocument( const string& aContent);
```

Note
Older versions of the documentation tools render an `@exception` as an @leave.

@panic

Purpose

Describes a panic condition of a function.

Syntax

Tag	Description
`@panic` *category number description*	Function will panic with *category number* in circumstances explained by *description*. *category*, *number* and *description* are required. *category* is the value of the string passed as the first parameter to a `User::Panic()` call and *number* is the value of the integer passed as the second parameter. *description* always consists of plain text line(s) terminated by either a blank line, or the next @ tag in the comment block, or the end of

	comment block delimiter.

Usage

Only panics categories defined within this interface need be documented.

A separate instance of this tag is required for each panic generated by the function.

Example

```
/**
Deletes the specified resource file from the list maintained by CCoeEnv.

@param aOffset  The offset value defined for the resource file to delete.
@panic CONE 13  The specified resource file does not exist in the list.
*/
EXPORT_C void CCoeEnv::DeleteResourceFile( TInt aOffset )
    {     ...     }
```

Informational

@see

Purpose

Use in source or header files to provide additional documentation context.

Syntax

Tag	Description
`@see` *`name`*	*`name`* is the name of another class, function, enum, etc. See Linking to Other API Items for details of the *`name`* format.

Usage

The `@see` tag generates a hyperlink in the documentation build, for any name recognised as a class, function, or other type in the system. When extracted, this tag creates a hyperlink to the extracted documentation for the named class or function.

To link to multiple items, use multiple `@see` tags. You cannot link to multiple items from a single tag.

It is recommended for use when an interface has a strong relationship to another interface, and this is not otherwise clear in the class or function declaration. For example, if a class is always used as a parameter in another class, `@see` is a concise way to help the user know this.

Two names joined together by `::` are understood as referring to a class and one of its members. One of several overloaded functions or constructors may be selected by including a parenthesized list of parameter types after the function name.

Example

```
@see TOpenFontSpec::SetBitmapType()
@see TFontStyle
```

Example of linking to overloaded functions

```
@see CCalInstanceView::FindInstanceL( RPointerArray<CCalInstance>&,
                                      CalCommon::TCalViewFilter,
                                      const CalCommon::TCalTimeRange& ) const
@see CCalInstanceView::FindInstanceL( const CCalFindInstanceSettings& ) const
@see CCalInstanceView::FindInstanceL( const CCalFindInstanceSettings&,
                                      TCalLocalUid, const TCalTime& ) const
```

@code and @endcode

Purpose

Provides a code block showing an example use of the interface.

Syntax

Tag	Description
`@code`	Marks the beginning of a block of text which should be treated as code, for example: with line-breaks preserved. Use with `@endcode`.
`@endcode`	Marks the end of a verbatim code block. This tag **must** be used to explicitly terminate the block. The `@endcode` tag **must** be by itself on a new line. If it is not, the tag will not be found, and all lines after it are also treated as part of the code block. Doxygen issues no warning about the 'missing' tag. Also note that if the `@endcode` tag is misspelled, the rest of the file will not be documented.

Usage

Code blocks must be enclosed between `@code` ... `@endcode` tags in order to be extracted verbatim.

For readability, code blocks may be separated from overview text by a blank line above and below.

Use **spaces** to indent code. Using a mixture of tabs and spaces to indent code will usually fail to align properly in the output. Code samples are an effective way of showing the

use of classes and functions, and their use is recommended for any item requiring complex explanation.

Example

The following example is from the description of the RSqlStatement class. It shows a recommended pattern for using a particular overload of `RSqlStatement::ColumnBinary()`, and shows the distinctive recommended approach to iterating through database records.

```
@code
RSqlDatabase database;
.........
RSqlStatement stmt;
TInt err = stmt.Prepare( database, "SELECT BinaryField FROM Tbl1" );
TInt columnIndex = stmt.ColumnIndex( "BinaryField" );
while(( err = stmt.Next() ) == KSqlAtRow )
    {
    TPtrC8 data;
    TInt err = stmt.ColumnBinary( columnIndex, data );
    if ( err == KErrNone )
        {
        // do something with the data         }
        }
    if (err == KSqlAtEnd )
        // OK - no more records
    else
        // Process the error
    stmt.Close();
    }
@endcode
```

Example

```
/**
...
@code
CRequestor* service = CRequestor::NewLC(
   CRequestor::ERequestorService,
   CRequestor::EFormatApplication,
   KSrvName);
stack.Append( service );
@endcode
...
```

Notes

The Doxygen tags `@verbatim` and `@endverbatim` are treated in the same way as `@code` and `@endcode`.

If using out-of-source comment files (commonly known as `.dosc` files) then Doxygen will respect C style comments within `@code ... @endcode` tags (normally the trailing `*/` would terminate the Doxygen comment). However there are a couple of quirks in Doxygen's behavior:

Doxygen v1.3.x throws away Doxygen-style C comments within a `@code ... @endcode` block. So in these instances use `/*...*/` rather than `/**...*/`.
Using `#define FOO /*...*/` within a `@code ... @endcode` block is seen by Doxygen v1.3.x as a `#define` declaration rather than a comment. So in these instances use `#define FOO //...` or put the C-style comment on a line prior to the `#define ...` line.

Platform Security

@capability and @ref

Purpose

Describes the platform security capabilities required to use a client-side entry-point function to an IPC server call.

Note that functions can have:

Capabilities that are **unconditionally** required, regardless of any parameters passed to them or the state of environment. Such capabilities are straightforwardly marked using the `@capability` tag with a named capability.
Capabilities that are **deferred** to other functions. Deferred capabilities allow a function to assume the capabilities of one or more functions. Deferred capabilities, once resolved, are equivalent to unconditional capabilities. If functions have deferred capabilities then the sum of that function's unconditional capabilities becomes that function's unconditional capabilities plus the sum of any deferred function's deferred capabilities. In other words, **deferred capabilities are aggregated sets of unconditional capabilities.** Functions that defer to target functions do not have any influence on the target function; deferring is *one-way*. Such capabilities are marked using the `@capability Deferred` with a reference to a named function.
Only functions can defer.
Functions can only defer to other functions.
Functions can defer to any number of other function(s).
Unreachable references are ignored.
Reference chains with cyclic references are ignored.
Deferred capabilities are additive.
Deferred capabilities are non-exclusive, meaning one capability does not preclude another.
Capabilities that are conditionally required, depending on the values of enumerated values or global variable passed to them as parameters. In this case, the `@capability` tag is put on the enumerated value or variable, and a `@ref` tag additionally used to specify the function to which the capability applies.

Syntax

Tag	Description
`@capability` *`name explanation`*	Capability *`name`* protects this function as explained in the *`explanation`*. Required where a function is protected by a capability. *`name`* is a capability from the `CapabilityNames` array (see tables below). *`explanation`* is optional and could be used to say what the capability is used to protect.
`@capability Deferred @ref` *`function_name`*	Capability is deferred to function *`function_name`*. See the use of `@ref` below.

`@ref` *`name text`*	This is used in conjunction with the `@capability` tag, if the capabilities of the function are conditional.
	
	`name` is the name of the function to which the capability check applies.
	
	For non-overloaded functions, the fully qualified name must be used. For example:
	`@ref Anamespace::CClass::CEmbeddedClass::MemberFunction`
	
	Or, for functions in the global namespace:
	`@ref GlobalFunction`
	
	For overloaded functions, the fully qualified name and all the types in the function declaration must be used, for example:
	`@ref Anamespace::CClass::CEmbeddedClass::MemberFunction(TInt, char*)`
	
	Or, for functions in the global namespace:
	`@ref GlobalFunction(char*)`

The tables below lists the choice of capability name specified in the `CapabilityNames` array (in `e32capability.h`). For explanations of these capabilities, see the documentation for the `TCapability` enum.

System capabilities				
NetworkServices	LocalServices	ReadUserData	WriteUserData	Location

User capabilities				
TCB	CommDD	PowerMgmt	MultimediaDD	ReadDeviceData
WriteDeviceData	DRM	TrustedUI	ProtServ	DiskAdmin
NetworkControl	AllFiles	SwEvent>	SurroundingsDD	UserEnvironment

Note
*There is **no** such capability as `@capability None`. If a function has no capabilities do not give it a capability tag.*

In addition to these capabilities, there are additional values that can be used for the name, which have special behavior:

`Dependent` This is used to indicate that a function that has no unconditional capabilities has conditional capabilities specified elsewhere. See the example Documenting functions that only have conditional capabilities.
`Note` This puts a comment in the capability section of the Developer Library for a function. Use it to document information about the function's capability in general (in other words, the documentation is not associated with a specific capability name). See the example in Documenting general capability information.
`Deferred` This is used to indicate that a function that has deferred capabilities. See the example Deferred capabilities.

Usage

Documenting security-related behavior is essential to enable clients to use controlled server interfaces. Every client-side entry point to an IPC server call that is controlled by one or more capabilities should be documented, using one `@capability` tag per capability. Name each capability, and explain what it protects.

Document only capabilities controlled directly by the entry point (that is – the function that is directly related to an IPC value).

Document the effect of having and not having the capability either in the explanation of @return or @leave, as appropriate.

Examples
Unconditional Capabilities

In this case, one or more capabilities are unconditionally assigned to functions. This is done by putting one or more `@capability` tags with unique capability names in the comment block describing the function.

The following code assigns `NetworkServices` and `LocalServices` capabilities to the function `Foo()`:

```
/**
Description of Foo.
@capability NetworkServices
@capability LocalServices
@return KErrPermissionDenied when insufficient capabilities
*/
TInt Foo( void );
```

Deferred Capabilities

A function can defer its capabilities to another function:

```
/**
@capability Deferred @ref Bar
*/
void Foo( void );

/**
@capability NetworkServices
*/
void Bar( void );
```

In this case, both `Foo()` and `Bar()` have `NetworkServices` capabilities.

The following code example has multiple functions and multiple capabilities:

```
/**
@capability Deferred @ref Eggs
*/
void Spam( void );

/**
@capability LocalServices
@capability Deferred @ref Chips
*/
void Eggs( void );

/**
@capability NetworkServices
*/
void Chips( void );
```

In this case:

`Spam()` has `LocalServices` from `Eggs()` and `NetworkServices` from `Chips()`.
`Eggs()` has `NetworkServices` from `Chips()`.
`Chips()` has `NetworkServices` unconditionally.

Conditional Capabilities

A function can have capabilities conditional to the values of enumerated values or variables. This is done by putting one or more `@capability` tags in the enumerated value or variable comment block with an `@ref` tag followed by the name of the function. One or more `@capability` tags with unique capability names can be used and one or more unique `@ref` tags can be used for each capability.

This variable confers `NetworkServices` on a couple of functions, for example:

```
/**
@capability NetworkServices
@ref Foo
@ref Bar
*/
const int IntNetworkServices = 11;

void Foo( int );
void Bar( int );
```

A more complicated example:

```
enum TMyEnum
    {
    /**
    @capability NetworkServices Stuff about the effect of enum_value_1 on Foo() and Spam()
    @ref Foo See note below for how to use this tag
    @ref Spam
    @capability LocalServices Stuff about the effect of enum_value_1 on Bar()
    @ref Bar
    */
    enum_value_1,
    /**
    NOTE: Foo, Bar reversed in this comment.
    @capability NetworkServices Stuff about the effect of enum_value_1 on Bar()
    @ref Bar
    @capability LocalServices Stuff about the effect of enum_value_1 on Foo()
    @ref Foo
    */
    enum_value_2,
    };

void Foo( TMyEnum aValue );
void Bar( TMyEnum aValue );
void Spam( TMyEnum aValue );
```

In this example, the following relationships can be asserted:

`enum_value_1` has a `NetworkServices` capability influence on `Foo()` and `Spam()` and a `LocalServices` capability influence on `Bar()`.
`enum_value_2` has a `LocalServices` capability influence on `Foo()` and a `NetworkServices` capability influence on `Bar()`.

Similarly we can infer the reverse assertions:

`Foo()` and `Spam()` are influenced by `enum_value_1` for the `NetworkServices` capability.
`Foo()` is influenced by `enum_value_2` for the `LocalServices` capability.
`Bar()` is influenced by `enum_value_1` for the `LocalServices` capability and by `enum_value_2` for the `NetworkServices` capability.
`Spam()` has no `LocalServices` capability.

Conditional and Unconditional Capabilities

Using the example from the Conditional capabilities section above, if `Spam()` is given an unconditional capability thus:

```
/**
@capability LocalServices
*/
void Spam( TMyEnum aValue );
```

then the relationships in the Conditional capabilities section hold with the addition that `Spam()` now has a `LocalServices` capability independent of `enum_value_1` or `enum_value_2`.

Documenting Functions that Only Have Conditional Capabilities

If a function only has conditional capabilities, it is convenient to allow a `@capability` tag to be applied at that function so that anyone looking at the source code will realize that it has conditional capabilities. So that the text following the `@capability` is not misunderstood as an unconditional capability, the keyword `Dependent` is reserved and used merely to

indicate that conditional capabilities are present.

For example, the function `Foo()` in the Conditional capabilities section could be documented thus:

```
/**
...
@capability Dependent Capability required depends on the value of aValue used
...
*/
void Foo( TMyEnum aValue );
```

Documenting General Capability Information

If it is desired to document information about a function capability in general (meaning that the documentation is not associated with a specific capability name), then the `Note` keyword can be used.

For example, the function `Foo()` in the Conditional capabilities section could be documented thus:

```
/**
...
@capability Note Some general comment here.
...
*/
void Foo( TMyEnum aValue );
```

The information will appear in the Developer Library.

Internationalization

This chapter gives recommendations and guidelines on how to create code that can be localized to various languages and regions most easily and efficiently, minimizing engineering changes.

SFCSR-320 Keep Code and Content Separate

One of the key issues in creating localizable software is separating material that needs to be localized from the software code. Material to be separated from the code includes, but is not limited to, the following:

All text items in menus, dialogs, notes and so on.
System-related items, such as folder names, icons and so on.
Any other text that is visible to the end user, such as strings in `.aif` files, skin names, and user-definable settings such as language or locale.
Text style, casing and font.

Place the localizable elements in separate resource header files (localization text file). Use **text ID**s to represent UI strings in the resource files, and place the definitions of the strings in corresponding localization text files.

SFCSR-321 Use Locales

A locale is a set of data values the vary according to language and geographic region. These can be utilized through the class `TLocale`. All Symbian OS developers should be familiar with the class, and it must be used where appropriate to produce language- and location-independent code.

There should not be any hard-coded language or country-dependent information such as date, time, currency, and so on in the program code.

The `User` class library also possesses static functions, some of which are language dependent. These include `User::Language()`, which returns the language of the current locale.

SFCSR-322 Date and Time Formatting

All the date and time formatting strings have to be included in the localization text file, because formatting the date and time in Symbian OS is not flexible enough for some languages. Therefore, in some cases, the formatting string needs to be changed to suit the conventions of a particular language at localization phase.

For example:

```
#define text_date_short_with_zero "%D%M%Y%1%/1%2%/2%3"
```

Use common components to format date/time data:

Call the class `StringLoader:Load` variant to load common date/time formatting string from the component `avkon`.
Use `TTime::FormatL` to format the date/time data according to the formatting string (and current locale).
Use `AknTextUtils::LanguageSpecificNumberConversion` to ensure correct number formatting in all languages before outputting a string.

SFCSR-323 Currency Formatting

Never assume the position of a currency symbol with reference to the number, number of characters for the currency symbol and how the negative currency is formatted. Example:

In the USA, you get $5.
In France, it is 5 €.

Use `TCurrencySymbol::Set()` to get the currency symbol and the formatting functions in `TLocale` to format a currency value.

SFCSR-324 Collation

Collation should be applied only to whole strings. In general, collation should use the `CompareC()` descriptor function and not pattern-matching functions based on single characters.

Symbian OS supports multiple collation methods within a locale, each of which has a unique identifier. Many locales have customized standard collation methods, and applications should not rely on collation methods being the same across all locales. The static function `Mem::CollationMethods()` can be used to get the number of collation methods supported by the current locale. In addition, within a collation method, Symbian OS supports collation levels that can be used to specify the exactness of matching required.

SFCSR-325 Decimal Separators

Never assume what character is used for decimal separator. Use `TLocale::DecimalSeparator()` to get the locale-dependent decimal separator.

SFCSR-326 Allow for Text Expansion

English is a relatively compact language. When an English-language string is localized, its length (and needed size of a buffer to store the value) is likely to increase approximately 30% to 50%. For this reason, UI items should be designed to allow the UI texts to expand. The shorter an English string is, the more it is likely to expand in another language. Allow 400% expansion for strings containing a short single word.

For example, when *to* is translated into Finnish it might become *vastaanottaja* (*recipient*), and *OK* is *aceptar* in Spanish.

The following table contains the recommended text-expansion percentages:

Number of characters in English	Text expansion %
1-4	400
5-10	100
11-20	70
21-30	50
31-50	30
Over 50	20

SFCSR-327 Use `TBUF`s for Localizable Strings

All localizable strings need to be in the resource file as `TBUF` resources.

If fixed-length `TBUFnn` resources are used in a resource file (nn is the size of the buffer, for example, `TBUF80`), buffers of the same length (`TBuf`) must always be allocated for these strings. Before you assign the buffer size, make sure you have a proper English text that makes sense (if the English text does not make sense, it is unlikely the localized strings will). Then, count the size of the string and allow for text expansion. This gives you a number that you will use when you pick a suitable `TBUFnn` resource variant.

SFCSR-328 Do Not Concatenate

Many programmers like to construct strings from parts in their code at run-time (also known as concatenation). This is only acceptable if the compound parts will not be localized; for example, if the string to be constructed contains a Web address.

Concatenation is not allowed for strings that need to be localized. Since all translatable text must be separated into the localization text files, the text fragments of the compound string will also end up in the file. These fragments will likely prove difficult or impossible to translate, not only because it is difficult to understand which parts belong together, but also because in most languages a word has genders or otherwise needs a different appearance depending on context.

SFCSR-329 Do Not Re-Use Strings

As a general recommendation, avoid re-using strings, because when re-used, the string may appear in a different **context** and may have a different meaning. Similarly, the **available space** may be different. As a result, it is recommended that you create different UI strings for all seemingly identical cases.

For example, you cannot re-use a string that has an adjective without a noun; for example 'New' in 'File New'.
Similarly, do not re-use strings between applications. This is because the same string can have a variety of meanings depending on the context, and it must be possible to create separate translations for each purpose. Also the space available for the translation may vary from application to application.
Create a separate string for each context and each occasion when the space available changes.

Exception: The strings declated in the in the `avkon.loc` can be used by other applications as long as the context and layout are identical (for example, soft key labels such as OK, Cancel, Close).

SFCSR-330 Use Re-Orderable Parameters in Strings

If there is only one parameter in a string, the parameter does not need to be ordered. If there are several variables in a string, their order may need to be changed in localization.

`StringLoader` class is a solution developed for loading and ordering parameters in strings and it should be used in all string cases, even those involving only one string. **Note:** Although you use parameters to handle variable parts, do not use parameters to concatenate texts.

SFCSR-331 Do Not Use Text in Graphics

Avoid using text in graphical UI elements. There may be instances where the graphics need to be re-drawn for new markets to take cultural requirements into account. If text is really needed in graphics, a better solution would be to use a text string on top of graphics.

SFCSR-332 Use Common Components

Use the `AVKON` component wherever possible. Most of the writing system and language support has been embedded in `AVKON` .
Use `AVKON` time and date formatter strings for date/time formatting, or if you need to have new formats, use locale formatters.
Use `AknUtils.h` and `TLocale:: DigitType()` to manipulate digits based on language setting.
Use the `StringLoader.h` for re-orderable parameters
Use `CharConv` for inbound and outbound character conversions.
Use locale (`TLocale`) to access locale sensitive data and sorting, for example, digits or date/time formats.
Use bi-directional utilities such as class `AknTextUtils` and `AknBidiTextUtils` in code that needs supporting languages that require conversion from logical to visual form, such as Arabic and Hebrew.

Indenting

SFCSR-340 Indenting

All C++ statements within a block of code should be indented exactly **four spaces** deeper than the enclosing block.
Configure your editor so that it uses **spaces** instead of tab characters, because different editors display tabs differently.

SFCSR-341 Placement of Braces and Parentheses

The placement of the **{** and **}** braces is the same level of indentation as the code they are enclosing. Note, that some editor tools can be configured to indent the code according to this recommendation.

Examples:

```
void CAknDiallerLauncher::StartKeyTimerL()
    {
    if ( !iKeyTimer )
        {
        iKeyTimer = CPeriodic::NewL( CActive::EPriorityStandard );
        }
    else if ( iKeyTimer->IsActive() )
        {
        iKeyTimer->Cancel();
        }
    iKeyTimer->Start( KAknKeyboardRepeatInitialDelay,
                      KAknKeyboardRepeatInitialDelay,
                      TCallBack( ReportLongPressL, this ) );
    }
```

SFCSR-342 Usage of Space
Primary operators are written with **no spaces** around them. Primary operators are:
`lvalue.id`
`var.*ptr`
`table[index]`
`pstruct->id`
`pstruct->*id`
Unary operators are written with **no spaces** between them and their operands.
Exceptions: **new** and **delete**
All other operators are written with **one space on each side** of the operator.
Parentheses have always a space on the inside, except for **()**.
There is no space between a function or macro name and its parenthesized parameter(s).
There should be no spaces between the **#** and the keyword.
There is be a single space between a keyword, for example, **if** or **for**, and its parenthesized argument(s).

Examples:

```
if ( iPointer->Function() > KOurMaxValue )
    {
    iCounter++;
    variable = iTable[a] + iOffset;
    }
```

```
for ( TInt i = 0; i < numMaps.Count(); i++ )
    {
    if ( numMaps[i].iChar == '*' )
        {
        starScanCode = numMaps[i].iKey;
        break;
        }
    }
```

```
iMember.Function1()
iMember.Function2( first, second );
```

Wrong:

```
// Spaces should appear inside the parenthesis.
Modify(parameter);

// There should be one space between "if" and "(".
if( error )
    {
    }
```

Parentheses should always be used where there is potential ambiguity.

SFCSR-343 Maximum Line Length
Prefer a maximum of 78 characters per line. Longer lines are hard to read and some editors cannot even handle them properly.
Split longer lines into several lines.

For example:

```
class CMessageListView:
   public CPknView,
   public MEikListBoxObserver,
   public MMsvSessionObserver
   {
public:
   }
```

or

```
void DoActivateL(
    const TVwsViewId& aPrevViewId,
    TUid aCustomMessageId,
    const TDesC8& aCustomMessage );
```

or

```
if ( isThisIntegerNumber > KSomeMinValue &&
     isThisIntegerNumber < KSomeMaxValue &&
     somethingElse >= 0 )
```

Casting

SFCSR-344 Typecasting

As always, casts should be used with caution. If a cast seems to be needed, you should first try to improve the design to avoid the need for a cast.

A cast must be C++ style. C-style casts and deprecated Symbian OS casting macros shall not be used.

Exception:

`dynamic_cast<>` operator should not be used as runtime type information (RTTI) it is not fully supported by Symbian OS (see further explanation below).

Note: For historical reasons, Symbian OS provides the following deprecated macros to encapsulate the C++ cast operators:

`REINTERPRET_CAST`

STATIC_CAST
CONST_CAST
MUTABLE_CAST

These shall not be used in new code. They were needed because some older compilers did not fully support C++ casts.

Explanation of dynamic_cast<>

The general implementation of dynamic_cast<> relies on the one-address-rule (OAR), which Symbian OS does not comply with in the general case due to having a proper C++ DLL model (rather than a C-based DLL model that allows C++ within the DLL). Symbian OS does support the one-definition-rule that is sufficient for dynamic_cast<> to work in almost all cases. So if your code meets the OAR then dynamic_cast<> will work. If it doesn't, then it won't.

What does this really mean?

dynamic_cast<> will work in Symbian C++ unless both the following conditions hold:

You are casting to a type whose RTTI is in a different DLL.
The class has no key-function; in other words, it only has pure virtual functions.

What do I do if it fails?

If you can modify the DLL, then you can add a function that is not pure virtual. If you can't modify the DLL, then there is no known work-around.

Exceptions

Symbian manages error flow in programs through leaving and panics. These two mechanisms must be used to ensure that all error cases are handled adequately.
C++ style exceptions are not fully supported in the current Symbian.
For further details of using C++ exceptions, see Hybrid Coding Guide.

Recurring Patterns

Two-Phase Construction

Complex objects (typically those deriving from CBase) require two-phase construction.

Trivial constructor
Non-trivial construction – allocation of resources
Object only fully initialized when both phases have completed
Static factory functions can wrap both phases into a single call.

Trivial Construction

SFCSR-350 C++ Default Constructor Cannot Leave

To avoid leaving in C++ default constructor

Use a member-initialization list, rather than assignment statements, to initialize member variables in a constructor.

Non-Trivial Construction

SFCSR-351 Use ConstructL

A class that allocates heap resources during construction must implement a ConstructL() function and a static factory function. **Example:**

```
// This completes initialization of the object
void CComplexObject::ConstructL()
    {
    iResource = new ( ELeave ) CResource;
    }
```

Static Factory Construction

SFCSR-352 A Static Factory Function is NewL() or NewLC()

Example:

```
/
// Return a handle to a new fully initialized instance of this class
//
CComplexObject* CComplexObject::NewL( TInt aLength, TInt aValue )
    {
    CComplexObject* self = new ( ELeave ) CComplexObject( aLength, aValue );

    CleanupStack::PushL( self );
    self->ConstructL();
    CleanupStack::Pop( self );

    return self;
    }
```

Assertions

Catch programming and run-time errors early by using pre- and post-conditions in functions.

Assert that those conditions required for correct execution hold true. Two mechanisms support this programming style.
__ASSERT_DEBUG
__ASSERT_ALWAYS
Class invariants.

SFCSR-353 Use __ASSERT_DEBUG

Use __ASSERT_DEBUG to catch programming errors. This panics client only in debug builds.
In general, generating User::Panic must be used with caution; the leave mechanism must be used instead whenever possible.
Use __ASSERT_ALWAYS with care because it also panics client in release mode. Comment __ASSERT_ALWAYS usage beside the call.

Class Invariants

For non-trivial classes use __DECLARE_TEST and __TEST_INVARIANT macros to ensure your objects are in a valid state where appropriate. These can be called at start or end of public methods of the class.

Explanation: Define class invariants for non-trivial classes using

__DECLARE_TEST – Specifies the allowed stable states for that class.

Call the invariant at the start of all public methods (where feasible) using

`__TEST_INVARIANT` – Ensures the object is in a stable state prior to executing the function. Calls are compiled out in release software builds.

For non-const methods call the invariant at the end of the method. This ensures the object has been left in a stable state after executing the method

For example:

```
//
// Removes text content, commencing at position aPos, over aLength number of characters
//
void CComplexTextObject::Delete( TInt aPos,TInt aLength )
    {
    __TEST_INVARIANT;
    __ASSERT_DEBUG( aPos > 0, Panic( EPosOutsideTextObject ) );
    __ASSERT_DEBUG( aLength >= 0, Panic( EDeleteNegativeLength ) );

    iTextBuffer->Delete( aPos, aLength );

    __TEST_INVARIANT;
    }
```

Cleanup Stack

The cleanup stack allows local pointer variables that are pushed to the stack to be cleaned up automatically in case a function leaves. You should generally prefer to use the cleanup stack instead of the TRAP harness. TRAP harnesses are expensive (slow, add complexity and code size, ~50 bytes per TRAP). Note that if you need to add an extra TRAP inside a leaving function, you need to take a look at your error design.

SFCSR-354 Use Cleanup Stack

The cleanup stack shall be used to ensure that a temporary CBase-derived object allocated on the heap is destroyed in the event of a leave.

Use the checking methods of `CleanupStack::Pop()` to check that `PushL()` and `Pop()` are balanced.

Use `CleanupStack::PopAndDestroy()` if you have finished using the object. **Example:**

```
CObject* object = CObject::NewL();
CleanupStack::PushL( object );
object->LeavingFunctionL();
CleanupStack::PopAndDestroy( object );
```

A CBase-derived class's virtual destructor is automatically invoked during cleanup, which is not the case for other classes. For other classes, you can provide explicit cleanup support via the `TCleanupItem` class.
Never push member data on the cleanup stack, as this will result in a double-deletion memory leak.
`Pop( reference )` should always be used in favor of `Pop()` or `Pop( integer )`. This will expose pop sequence error bugs early, through a crash, instead of some strange behavior that might be more difficult to debug.

RClasses

SFCSR-355 Use `CleanupXxxxPushL()` Method for R-class

Use `CleanupXxxxPushL()` to ensure that a temporary R-class object is cleaned properly in the event of a leave.

Example:

```
RObject object;
object.Open();
CleanupClosePushL( object );
object.LeavingFunctionL();
CleanupStack::PopAndDestroy( &object );
```

Note: This may not be suitable for all R-class objects.

If you call non-leaving functions after `LeavingFunctionL` it is more efficient to leave the object on the cleanup stack and to call `CleanupStack::PopAndDestroy()` afterwards.

TCleanupItem

SFCSR-356 Use `TCleanupItem` for Non-Trivial Cleanup

`TCleanupItem` shall be used to ensure that an object requiring non-trivial cleanup is cleaned up in the event of a leave.
Some classes or methods require more complicated cleanup. Use a `TCleanupItem` and provide a local or static function that does the required cleanup.

Example:

```
// Allows correct cleanup of specified objects by
// using the cleanup stack instead of a TRAP harness.
void RObject::ReleaseOnCleanup( TAny* aObject )
    {
    reinterpret_cast< RObject* > ( aObject )->Release();
    }
RObject* object = GetObject(); // may increase share on a reference counted object
CleanupStack::PushL( TCleanupItem( ReleaseOnCleanup, object ) );
iContainer->AddL( object );
CleanupStack::Pop( object ); // Pop the TCleanupItem::iPtr
```

Heap Allocated Arrays

SFCSR-357 Use `CleanupArrayDeletePushL()` for Array

The `CleanupArrayDeletePushL()` method must be used to ensure proper cleanup of an array in the event of a leave.

This ensures that `PopAndDestroy()` will call `delete[]` to delete the array correctly.

Example:

```
TObject objectArray = new ( ELeave ) TObject[numberOfElements];
CleanupArrayDeletePushL( objectArray );
LeavingFunctionL();
CleanupStack::Pop( objectArray );
```

Private Inheritance

SFCSR-358 Private Inheritance is Forbidden

Use composition instead. Inheritance should be restricted to cases when the relationship really is a 'kind of'.

Multiple Inheritance

SFCSR-359 M Class Must Not Include Implementation

M class (mixin) must specify interface only using pure virtual functions.

M class has no implementation.
M class has no member data.
M classes may only be derived from other M classes.

SFCSR-360 A Class Must Inherit From Exactly One C Class

A class must inherit from only one CBase-derived class. Additionally, the class may also inherit from any number of M classes (mixins). The C class must be the first listed.

Multiple inheritance is fine provided only multiple **M** classes (mixins) are inherited.
No M class may be mixed in more than once in any class, either as a direct base or as a base of any of its primary base classes.
Inherit from a CBase-derived class first, to achieve correct layout of the v-table.

Example:

```
class CGlobalText : public CPlainText, public MLayDoc, public MFormatText
    {
    };
```

```
class CDerivedClass : public CBase, public MSomeObserver,
    public MSomeOtherObserver
```

`User::Panic`

SFCSR-361 `User::Panic` Must Be Used With Caution

The leave mechanism must be used instead whenever possible.

`User::Panic` should only be used in R&D builds within the `_ASSERT_DEBUG` macro.
If you must use it in other situations as well (`_ASSERT_ALWAYS`), you must have very good reasons for it and comment it beside the call. For example, it is better to panic a client (usually an application) instead of passing clearly malformed data to a system-vital service such as servers controlling all the functionality of all applications.

Data Types

Constants

SFCSR-370 No Magic Constants

Never use hard-coded constants, such as 'magic numbers', in your code. Use **enums**, resources and real **const** definitions instead. For example:

```
switch ( aCommand )
    {
    case ETetrisDropBlock:
        ...
        break;
    case EEikCmdExit:
        ...
        break;
    default:
        // Handle unexpected CommandID here.
        ...
    }
```

Avoid preprocessor macros (**#define**s).
Collect your **const** and **enum** definitions in a single place so that they can be easily spotted for changes.
If the interface of the class itself uses enum values (for example, as default arguments), the enum values must be defined before they are used in a function declaration.
Use platform-defined macros and run-time library functions instead of hard-coded directory paths.
Use resource IDs of localized texts instead of hard-coded literals.
Use Central Repository and Publish and Subscribe constants instead of copying the same values to your code.
Never define a **string constant**. Use a resource file instead. As an exception, string constants may be used where they do not appear on the UI and are not language dependent: for example, protocol strings, resource names, system file names.
Never define a **color constant** of your own. Use a predefined **library constant** instead.

Example:

```
const TRgb KTetrisBgColor( KRgbBlack );
...
TRgb backgroundColor( KTetrisBgColor );
```

In this way, you ensure your code remains future-proofed against any code changes that may be made later on elsewhere in the system.

Variables

SFCSR-371 Initialize Variables

Initialize variables when you create them.

Use **initialization** instead of **assignment** when it is more effective.
Never initialize or instantiate multiple items on the same line.
Do not declare **automatic** variables until required
Large automatics (for example, `TBuf<1024>`) should not be used. Those could lead to stack overflows. Allocate on the heap instead.
Initialize a pointer to NULL, not to zero.

Example:

```
TBool isChecked( EFalse );
TInt i( 0 );
TRect rectangle( 0, 0, 0, 0 );
CTetrisClass* pointer = NULL;
TPtrC version( KNullDesC );
TInt domTreeHandle( KNullHandle );
```

SFCSR-372 No Static Variables

Never use **static** local variables.
Dynamic link libraries (DLLs) should not have any global non-const data. This includes static member variables.
Symbian's Real-Time Kernel (EKA2) introduced support for DLL writable static data (WSD). However, this must be used carefully and the limitations and costs need to be well understood. For example, you cannot load multiple instances of the DLL in the emulator.

```
/**
 * Tetris game base class.
 */
class CTetrisClass: public CBase
    {
    ...
private:
    TInt iIntegerMember;
    COtherClass* iPointerToAnotherObject;
    ...
```

Wrong:

```
static int a,b,c;
// Static local variables are not allowed, TInt type must be
// used, variables should be initialized (to zero, for example)
// and multiple items should not be declared on the same line
```

```
class CBadClass
    {
    ...
    // This member function is OK:
    EXPORT_C static TInt MemberFunction();
    ...
    // But this data member is not:
    static TInt iStaticMember;
    // DLL's cannot have any global non-const data.
```

Types

SFCSR-373 Use Symbian Types

Never use primitive types (such as **int**, **char**, **float**, **double** or **bool**) directly.
Instead, use concrete types provided by Symbian OS.
Use descriptors (`TDes, TPtr`) and buffers (`HBuf, TBuf`) instead of ordinary pointers when storing data and handling strings.
Boolean type is `TBool` and it has values **ETrue** or **EFalse**

```
TBool booleanVariable( EFalse );
TInt pos( 0 );

TInt TAknTextWrapper::GetLocVariants( const TDesC& aText )
    {
    TPtrC remaining( aText );
    ...
    }
TDesC* titleText = iEikonEnv->AllocReadResourceLC( R_LEAP_SETTINGS_TITLE );
```

To compare Boolean (`TBool`) value in **if**() statement you should never compare it to any exact value.
Use the way shown in the example.

Example:

```
...
if ( booleanVariable1 ) // is it true ?

...
if ( !booleanVariable2 ) // is it false?
...
```

Pointers

SFCSR-374 Never Return Pointer to Local Data

Never return a pointer or a reference to any local data. Wrong:

```
TInt& AddFive( TInt aNumber )
    {
    TInt result( aNumber + 5 );
    return result; // Whoops, local reference returned
    }
```

SFCSR-375 Usage of NULL for Pointers

Initialize a pointer to NULL, not to zero.

Example:

```
CTetrisClass* tetrisPointer = NULL;
```

In **if**() statement you should never compare any pointer to any exact value, except for **NULL**.

```
...
if ( tetrisPointer != NULL ) // Check if it points to an object
...
if ( tetrisPointer == NULL ) // Check if it NULL
```

Assign or initialize a pointer to NULL every time when it does not point to any existing object.
After deleting a heap variable, assign NULL to the variable to prevent later attempts to delete the non-existing object.
In a destructor, it is not necessary to nullify a deleted member pointer.

Example:

```
CMyClass::~CMyClass() // destructor
    {
    delete iAnotherClass;
    }
CMyClass::FinishGame()
    {
    ...
    delete iTetrisPointer;
    iTetrisPointer = NULL; // this is important
    ...
    }
```

Objects and Classes

Class Behavior

SFCSR-380 C-Class Constructor

Every **C** class must have a private or protected constructor. The constructor (one- or two-phased) is never virtual. **Note:** the C++ default constructor must not contain any code that can leave. Use a member-initialization list, rather than assignment statements, to initialize member variables in a constructor.

SFCSR-381 Two-Phase Construction

Every class that owns any pointers requires two-phase construction.

For example:

```
class CComplexClass : public CBase
    {
public:
    ...
    static CComplexClass* NewL();
    static CComplexClass* NewLC();

private:
    CComplexClass();   # Private default constructor
    void ConstructL(); # Two-phase contructor
    ...
```

SFCSR-382 Copy Constructor

It is highly recommended that you make the copy constructor private in order to prevent the accidental copying of objects. This is not necessary for C classes because CBase already prevents copy constructing. Instead, you should explicitly use the constructor or *clone* method to make a copy of an object.

```
class CComplexClass : public CBase
    {
public:
    ...
    static CComplexClass* NewL();
    static CComplexClass* NewLC();

    // Make copying explicitly visible
    static CComplexClass* NewLC( const CComplexClass& aObjectToCopy );

private:
    CComplexClass();
    void ConstructL();

    // Disable copy constructor
    CComplexClass( const CComplexClass& aObjectToCopy );
    ...
    };
```

Direct initialization for class types must be used rather than copy initialization.

For example:

```
TParsePtrC parser( TPtrC( aFileName ) );
```

instead of

```
TParsePtrC parser = TPtrC( aFileName );
```

Here, the initialization of `TParsePtrC` requires

a call to `TParsePtrC::TParsePtrC( const TPtrC& )` followed by
a call to `TParsePtrC`'s copy constructor `TParsePtrC::TParsePtrC( const TParsePtrC& )`.

This is significantly more expensive than initializing the class type directly, which eliminates the redundant call of the copy constructor.

SFCSR-383 Class Destructor

The destructor of a C class and M class is always **virtual**.

SFCSR-384 Deleting a Heap Variable

Every time you allocate an object, you must ensure that it is deleted correctly. After deleting a heap variable, `NULL` must be assigned to the variable to prevent later attempts to delete the non-existing object. Note: It is not necessary to nullify a deleted member pointer in a destructor.

Usually the same program entity (such as class) that allocates memory on the heap also releases it. If some other entity releases the memory, it should be commented in the header file. Ownership can be passed to some other entity through a function call but, in that case, the transfer of ownership must be clearly indicated in the comments.

Example:

```
CMyClass::~CMyClass()
    {
    delete iAnotherClass;
    }

CMyClass::SomeFunction()
    {
    ...
    delete iAnotherClass;
    iAnotherClass = NULL;
    ...
    }
```

SFCSR-385 Global Static Data in Constructors or Destructors

Do not use any global or static data in constructors or destructors.

SFCSR-386 Assignment Operator

Do not write an assignment operator for C class, because the assignment of a compound class may require memory allocation.

Guidelines for Class Design

Use explicit access specifiers (`public`, `protected`, `private`) at the start of each class section.
Public data members should not be used, except where absolutely required.
A common use is in the linked-list implementations (`TDblQueLink`) provided by Symbian. Generally, private class members and public accessor functions should be used.
Interface member functions (public functions) should usually return a const reference or pointer to a member variable.
Returning handles to member data limits object encapsulation, as the internal class layout is somewhat visible through the interface.
Avoid using friend functions or classes between components, because it breaks the encapsulation.
If you return a pointer or a reference, remember that the lifetime of the object being returned must always extend beyond the scope of the function!
So do not return references to automatic variables, for instance.
Parameters of class type must be passed to functions by reference or pointer, not by value.
In the public API, pointer parameters imply passing ownership, and reference parameters imply usage only.
However, the passing of ownership must also be explicitly noted in the comments of the function that ownership is passed through.
Do not implement member functions in the class declaration, unless it's the null implementation ({}).
Use INL file for inline implementations
Ensure that variables are given the minimal lifetime necessary to perform their task, and do not use unnecessary variables.
Never push member data on the cleanup stack, because this will result in a double-deletion memory leak.
Never redefine non-virtual functions in a derived class.
Never redefine default parameter values in a derived class.

SFCSR-387 Use Access-Control Specifiers

Access-control specifiers must be included in a class definition.

SFCSR-388 Classes as Member Data

When declaring a C class as member data within another class, declare it as a pointer to the C class.

For example:

```
class CExample : public CBase
    {
    ...
private:
    CMemberData* iMemberData;
    };
```

rather than

```
class CExample : public CBase
    {
    ...
private:
    CMemberData iMemberData;
    };
```

This way, `CExample` is protected from binary incompatibilities caused by changes in size of `CMemberData`.

Function Parameters and Return Values

SFCSR-389 Read-Only Parameter or Return Value

A read-only parameter or return value must be **const**.

For example:

```
TInt MatrixWidth( const CHugeMatrix& aParam );
const CHugeMatrix& Matrix() const;
```

Try to use the least-derived class possible as a parameter. Use `void Foo( const CArrayFix& aX );` rather than `void Foo( const CArrayFixFlat& aX );`.

SFCSR-390 Passing Basic Types

A basic data type or small T Class (smaller than or equal to 8 bytes) may be passed by value.

Avoid using const when passing by value.

SFCSR-391 Passing Large Objects

An object larger than 8 bytes must be passed by reference or pointer.

Pass a pointer when transferring ownership of an object; pass a reference otherwise.
If a `NULL` (non-existent) object is meaningful (such that `const CType&` cannot be used) consider a function overload with one function taking a **const** reference and the other taking no parameter.
Parameters of class type must be passed to functions by reference or pointer, not by value.

Exporting

SFCSR-392 Exporting Functions

When writing a Symbian OS library, attention must be given to which functions are defined as `EXPORT_C`.

Use the `NONSHARABLE_CLASS` macro for classes that are internal to DLL and not meant not to be used outside DLL.
Use the `NONSHARABLE_CLASS` macro if a class is meant to be used outside DLL 'as it is' via explicitly exported functions. (Neither virtual table nor type information will be exported outside DLL, thus deriving from the class is impossible.)
Note that `dynamic_cast` for that class also stops working outside DLL.
Do not export any function that is not designed to be used outside of the DLL.

```
// Internal class
NONSHARABLE_CLASS( CAknIndicatorContainerExtension ) :
    public CBase,
    public MAknPictographAnimatorCallBack

// Use via exported functions only
NONSHARABLE_CLASS( CAiwServiceHandler ) : public CBase
    {
public:
```

SFCSR-393 Exporting Data Members

A data member must not be exported. An exported function may be used to provide a reference.

SFCSR-394 Exporting Private Functions

A private function may be `EXPORT`ed only

if it is accessed by a public inline function or
it is virtual and the class is intended for derivation outside the library.

SFCSR-395 Exporting Pure Virtual Functions

A pure virtual function must not be exported. (This does not apply to a virtual destructor.)

SFCSR-396 Exporting Inline Functions

An inline function must not be exported.

Virtual Functions

SFCSR-397 Virtual Member Functions

The destructor of a C class is always virtual. The constructor of a C class (one or two phased) is never virtual. Virtual functions implemented in a derived class must be grouped in the declaration according to their original base class with a comment detailing their origin.

It is not necessary to specify the virtual keyword for these functions.

A virtual function must not be inline. (This does not apply to a virtual destructor.)

Read more about usage of the virtual member functions here.

Inline Functions

SFCSR-398 Use Inline Keyword

The inline keyword must be explicitly specified for an inline function.

SFCSR-399 Implementation of Inline Functions

An inline function must not be implemented in a class declaration.

It may be implemented either at the bottom of the header file or in a separate `.inl` file, which must be included at the bottom of the header file. Inline functions must be small. Typically, they just get or set a member variable whose size is one or two machine words.

Control Structures

SFCSR-410 Switch Default Case

Every switch statement must have a `default:` clause at least for detecting unexpected switch expressions.
Use logging code or some other handler to catch unexpected switch expressions.
Case branches without a break statement should be avoided. If one has to be used, the flow through must be explicitly indicated by a comment.

Example:

```
switch ( aCommand )
    {
    case ETetrisDropBlock:
        ...
        break;
    case EEikCmdExit:
        ...
        break;
    default:
        // Handle unexpected CommandID here.
        ...
    }
```

SFCSR-411 Braces in Control Flow Statements

Always use compound statement braces ('{}') in all the control flow statements even if it is only a single statement or an empty block:
if – else
while
do – while
for

Example:

```
if ( myVariable )
    {
    DoSomething();
    }

if ( myTest )
    {
    DoTest1();
    DoTest2();
    }
```

If **switch** - **case** uses local variables, you should use begin and end braces.

Example:

```
switch ( aCommand )
    {
    case ECommandView:
        ...
        break;

    case ECommandPrint:
        {
        TInt printer( 0 );
        ...
        break;
        }

    default:
        // Handle unexpected command here.
        ...
    }
```

SFCSR-412 Control Flow Statements To Avoid

Avoid using the following control flow statements

goto
break
continue

Exception: **break** is used in case statements.

C++ Templates

Avoid defining your own templates. If you have to, the recommended way is to use Thin templates (http://developer.symbian.org/sfdl/doc_source/guide/EssentialIdioms/ThinTemplates.guide.html#idioms.thintemplates) to ensure the smallest possible code size while still gaining the benefits of C++ templates.

Keep in mind that different C++ compilers have different problems with templates.

Namespaces

SFCSR-420 Namespace Prefix

Namespaces may be used. However, all other conventions in this document related to naming of files, classes, variables and such are still valid.

The namespace name should contain the **component** or **collection** name as a prefix.

Note! Anonymous (*aka* unnamed) namespaces must not be used in header, example:

```
namespace //Wrong. Unnamed namespace must not be used.
{
//code here
}
```

SFCSR-421 Using Namespaces

Use **using** declaration to access namespace members.

Example:

```
using namespace  Anamespace;
Class::Function(…) //  OK
```

instead of

```
Anamespace::Class::Function(….)  // Wrong
```

The exceptions can be where you need to call multiple functions inside a namespace.

Miscellaneous

Copyright

As described in Coding Standards and Conventions/Programming Practices/Commenting, each source file shall contain the standard header described in this page.

Generic File Header

The generic header in **C** comment format from which the SFL and EPL headers have been derived is as follows:

```
/*
* Copyright (c) {Year(s)} {Company}.
* All rights reserved.
* This component and the accompanying materials are made available
* under the terms of the License "{License}}"
* which accompanies  this distribution, and is available
* at the URL "{LicenseUrl}".
*
* Initial Contributors:
* {Name} {Company} - Initial contribution
*
* Contributors:
* {Name} {Company} - {Description of contribution}
*
* Description:
* {Description of the file}
*
*/
```

where

{License}
The allowed values for the tag are:

SFL: **Symbian Foundation License v1.0** EPL: **Eclipse Public License v1.0** Note, that the text is between " characters.

{LicenseUrl}
The allowed values for the tag are:
SFL : http://developer.symbian.org/main/legal/Symbian_Foundation_License.pdf
(NOTE that previously http://www.symbianfoundation.org/legal/sfl-v10.html was used - this is now redirected to the new URL specified above)
EPL : http://www.eclipse.org/legal/epl-v10.html.
Note, that the text is between " characters.
License
Is the keyword reserved for tools in when the license changes from the SFL to the EPL. Enclose the license text with " characters.
{Year(s)}
The old value from the old header. If you need to modify source code later, add the modification year to the header.
{Name}
The name of a contributor, recommended format *Firstname Surname*.
{Company}
The name of company.
Description
Description text.
Where a contributor adds code to an existing file, the contributor should fill out the following in copyright header
Contributors:
{Name} {Company) - {Description of contribution}

Where a contributor adds new file, the contributor must add a copyright header using the same license as the rest of the package and fill out

Inital Contributors:
{Name} {Company) - Initial Contribution

SFL Licensed File Header

```
/*
* Copyright (c) {Year(s)} {Company}.
* All rights reserved.
* This component and the accompanying materials are made available
* under the terms of the License "Symbian Foundation License v1.0"
* which accompanies this distribution, and is available
* at the URL "http://www.symbianfoundation.org/legal/sfl-v10.html".
*
* Initial Contributors:
* {Name} {Company} – Initial contribution
*
* Contributors:
* {Name} {Company} – {{Description of contribution}}
*
* Description:
* {{Description of the file}}
*
*/
```

EPL Licensed File Header

```
/*
* Copyright (c) {Year(s)} {Company}.
* All rights reserved.
* This component and the accompanying materials are made available
* under the terms of the License "Eclipse Public License v1.0"
* which accompanies this distribution, and is available
* at the URL "http://www.eclipse.org/legal/epl-v10.html".
*
* Initial Contributors:
* {Name} {Company} - Initial contribution
*
* Contributors:
* {Name} {Company} - {Description of contribution}
*
* Description:
* {Description of the file}
*
*/
```

File Header Templates

Some templates in various formats are described below. For other comment syntaxes, please replace the comment characters.

```
/*
* Copyright (c) {Year(s)} {Company}.
* All rights reserved.
* This component and the accompanying materials are made available
* under the terms of the License "Symbian Foundation License v1.0"
* which accompanies this distribution, and is available
* at the URL "http://www.symbianfoundation.org/legal/sfl-v10.html".
*
* Initial Contributors:
* {Name} {Company} - Initial contribution
*
* Contributors:
* {Name} {Company} - {Description of contribution}
*
* Description:
* {Description of the file}
*
*/
```

Using C++ comment syntax:

```
//
// Copyright (c) {Year(s)} {Company}.
// All rights reserved.
// This component and the accompanying materials are made available
// under the terms of the License "Symbian Foundation License v1.0"
// which accompanies this distribution, and is available
// at the URL "http://www.symbianfoundation.org/legal/sfl-v10.html".
//
// Initial Contributors:
// {Name} {Company} - Initial contribution
//
// Contributors:
// {Name} {Company} - {Description of contribution}
//
// Description:
// {Description of the file}
//
```

In SIS packages:

```
;
; Copyright (c) {Year(s)} {Company}.
; All rights reserved.
; This component and the accompanying materials are made available
; under the terms of the License "Symbian Foundation License v1.0"
; which accompanies this distribution, and is available
; at the URL "http://www.symbianfoundation.org/legal/sfl-v10.html".
;
; Initial Contributors:
; {Name} {Company} - Initial contribution
;
; Contributors:
; {Name} {Company} - {Description of contribution}
;
; Description:
; {Description of the file}
;
```

In DOS command files:

```
@rem
@rem Copyright (c) {Year(s)} {Company}.
@rem All rights reserved.
@rem This component and the accompanying materials are made available
@rem under the terms of the License "Symbian Foundation License v1.0"
@rem which accompanies this distribution, and is available
@rem at the URL "http://www.symbianfoundation.org/legal/sfl-v10.html".
@rem
@rem Initial Contributors:
@rem {Name} {Company} - Initial contribution
@rem
@rem Contributors:
@rem {Name} {Company} - {Description of contribution}
@rem
@rem Description:
@rem {Description of the file}
@rem
```

Package Specific

This page can be used by package owners to add package-specific coding standards and conventions to the general **coding standards** documentation.

About these recommendations

They are not maintained by the Architecture Council. Feel free to extend them as you consider necessary as a package owner, but keep in mind the general recommendations specified by the **coding standards** and avoid conflicting with them. If there is any uncertainty, please raise questions on the discussion forums (http://developer.symbian.org/forum/) .

Package specific root Wiki page
It is named a like **Coding Standards and Conventions/Package Specific**/`packagename`, where
`packagename` is the name of the package from the system model. See Symbian_System_Model#List_of_Packages
Under the **Coding Standards and Conventions/Package Specific**/`packagename` it is recommended to follow the chapter and section names names of the **coding standards**. It is easier to merge some package specific add-ons to the general one via the Symbian_Foundation_Council_Charters#Architecture_Council_Charter.
Package specific Recommendation ID
It is prefixed with **SFCSR-**`Number` where
Number refers to starting from **1100 onwards** (the **coding standards** uses numbers between 1..999).
Range of 100 items per package should be enough, so first package reserves and uses range 1100..1199, second package 1200..1299, third 1300..1399, etc.
Retrieved from "http://developer.symbian.org/wiki/index.php/Coding_Standards_and_Conventions"
Categories: Coding Standards | Symbian C++ | Package Owners