# Compatibility Overview

## Source compatibility (SC) break

A source compatibility break occurs when an interface (API) used by an application/component is changed in such a way that the source code of that application/component must be modified to comply with the changed or replaced new API. Also recompiling/rebuilding of the modified source code is needed. The other way round, if the interface is source compatible, only recompiling/rebuilding is needed, without modifications to the source code using the interface. Because source-level interfaces are defined in the published header files, it is easy to see whether changes are source compatible by examining the changes made to header files. It is also possible to find SC breaks by simply compiling the source code; any SC breaks will show up as compiler errors. Source compatibility has been broadly maintained across Symbian OS and S60 platform versions.

### What causes SC breaks?

With the release of a new platform version and new services designed to replace functionality on the old platform, old interfaces cannot always be supported. In such circumstances, source compatibility is broken and applications using old interfaces must be reimplemented to use new interfaces instead, before they can be built for the new platform version. The most common case where a source compatibility break will occur is an interface that has already been marked as deprecated and which is finally removed.

## Binary compatibility (BC) break

A binary compatibility break means that an application that has been compiled/build with an SDK for an earlier platform version does not run (or work correctly) anymore in the new platform version and devices based on it. At least recompiling/rebuilding is needed, but a BC break often implies also an SC break, in which case also the source code must be modified. SC and BC are, in fact, orthogonal – you can have one or the other, neither, or both. Typically BC implies also SC, but in practice BC is preserved more actively than SC because BC is very important for end users running applications on their devices. There are situations where source compatibility is intentionally broken, but binary compatibility is still maintained by supporting deprecated APIs. It is also possible (but rare) that an interface that has not changed, might work differently and thereby return something unexpected and different when compared to the earlier version. In other words, it both works and compiles, but it may work differently, which is hard to spot. This is called **functional** or **logical break**. In practice, this can be considered a rare special case of a binary compatibility break.

### Full Binary Compatibility

Full binary compatibility describes the situation where an interface has not changed at all. Applications built against the interface will also work with older versions of that library. Typically only backward compatibility is maintained, and full BC is rare. In a case where both source and binary compatibility are preserved, and the application compiles and runs as expected, the term **functional compatibility** can also be used.

### Backward Binary Compatibility

Backward binary compatibility describes the situation where an interface has changed between versions — but it is still guaranteed that code compiled for an older release will run on newer versions without rebuilding.

### What causes BC breaks?

There are significant changes between different S60 platform versions, both to the underlying Symbian OS and also some S60 platform–specific interfaces. Backward binary compatibility is being maintained by supporting deprecated interfaces and implementing wrapper mechanisms whenever possible. However, this means duplicate implementations for certain components, and an increased number of DLLs in the platform. Given the strict memory constraints of a mobile device, it is not possible to support all deprecated components. Generally, if applications have been implemented using common high-level APIs, they will work in a new platform release without problems. Only applications that directly call low-level APIs when platform compatibility breaks occur, risk failing when running on the new platform; these will need to be reimplemented to use the new interfaces instead.

## Data compatibility

Data compatibility is compatibility between data created by one version of a component and another version of the component. This may be settings information (such as an .ini) file or application data that a component is managing.

## Deprecated interfaces

Binary compatibility has been broadly maintained between different S60 platform versions. When an interface change in the new platform version would introduce a BC break, backward compatibility is maintained by allowing new and old components to coexist, and by using a so-called wrapping mechanism for certain components, where old interfaces can be used to access new functionality. These old interfaces are then marked as deprecated, meaning that even though they are supported on this platform version, they are likely to be dropped in the next version.

## Wrappers


(C) Copyright Nokia 2013. All rights reserved.

Wrappers are implemented when an interface has been redesigned or replaced, but compatibility is still required with the old interface. In general, wrappers just redirect calls to equivalent functionality in the new implementation, transparently to clients. Wrappers can provide source compatibility, where the original header files are still provided, or binary compatibility, where equivalent DLLs are provided, or both. In some cases where functionality is deprecated, binary compatibility is provided via wrapper DLLs, but legacy header files are removed. This means that existing applications will still work correctly, but any new applications must use the new APIs.

## Intentional and unintentional breaks

While most of the breaks are intentional and planned, there may, occasionally, also be breaks caused by defects in the platform code that have been detected too late, either causing a source, binary, or functional break, or all of those.

## Device-specific issues

Besides possible issues and bugs in the S60 platform or the underlying Symbian OS, there are typically some device-specific issues, changes, or bugs affecting the compatibility. So even though the platform as such would be compatible, specific problems on certain devices can affect the compatibility, preventing the software to automatically run correctly on all devices where it is expected to. There are many hardware-related issues especially on the device level. Most device-specific issues, in particular intentional hardware-related issues such as the display resolution and color depth or availability of Bluetooth and camera, are not considered breaks, but something developers need to take into account in the application development.

## S60 licensee dimension

Typically the first device of a platform version is regarded as the reference implementation to develop for and to test against. This mostly works and applies quite well to one licensee's devices, that is, Nokia devices. However, it should be kept in mind that there are also devices from other manufacturers. Some S60 functionality is configurable, or there may be some implementation-specific issues causing variance between different licensees' devices. Other manufacturers may also innovate and differentiate their products from the corresponding Nokia lead devices, which causes variance within the platform. Even though the first Nokia device is often perceived as the "platform" by developers, such an assumption is false and will typically cause problems when running an application on all S60 devices. Similarly, because there may be some unintentional breaks or problems in the Nokia implementation, there may also be some unintentional breaks or problems in the implementations of other S60 licensees. At least the following areas have been found to cause problems:

- Different hardware characteristics, affecting, for example, performance.
- Different approaches to deprecation (removal of deprecated APIs), for example, there may be difference in which phase certain deprecated APIs (DLLs) are removed from the actual devices. Using deprecated APIs is not recommended.
- Some sensitive frameworks, such as the start-up sequence, which may be different on licensee devices when compared to the Nokia environment. Assuming a fixed start-up sequence based on how it is implemented in Nokia devices is false, and may cause problems when running the application on licensees' S60 devices.
- Directory path names. Use PathInfo to determine directory path names at run time.

## Public and non-public APIs

Developers should always use public (SDK) APIs, because compatibility is only guaranteed and preserved for the public APIs. For most developers this is not a relevant issue at all, because public APIs are anyway the only (as the name also suggests) publicly available APIs; whereas non-public APIs are not meant, designed, productized, or documented for developer usage, and compatibility is not guaranteed or preserved. That is why those other APIs are referred as "non-public APIs." In other words, those APIs are non-public for a reason, not because somebody would intentionally try to limit the possibilities of developers. Even if a developer would somehow be able get access to non-public APIs, using them is likely to limit the range of devices the code will operate successfully on, thus minimizing the marketability of the code.

## Internal links

- Preventing Compatibility Breaks


(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.