

Contents

- [1 Build Process](#)
 - ◆ [1.1 Need for Freeze](#)
 - ◆ [1.2 Reason for Freeze](#)
 - ◆ [1.3 Freeze Process](#)
 - ◇ [1.3.1 Example](#)

Build Process

Symbian application development best practice suggests keeping the main engine and gui part separately. A good design should avoid creating unnecessary dependencies between components. Also, maximizing code reuse so that common functional requirements can be satisfied from shared libraries is very desirable. The engine for the intended application should be a functional unit that stands apart from whatever GUI layer represents the data to the user and allows them to interact with it. For the purposes of flexibility, it is best to allow the engine to reside in its own DLL and to offer its own API to any UI layer that might want to use it.

Refer

Developing Series 60 Applications: A Guide for Symbian OS C++ Developers

Need for Freeze

DLLs should freeze their exports before release, so as to ensure the backward compatibility of new releases of a library. It works by ensuring that the function ordinals currently available in a released import library correspond with the ordinals used by future versions of the DLL, even when new functions are added.

Reason for Freeze

The [petran](#) tool strips a Portable Executable format file of its irrelevant symbol information for an ARM target; thus making DLLs much smaller. As a consequence, ARM targets only support linking by ordinal.

Freeze Process

When the DLL is built each exported function is assigned a unique integer value known as an ordinal. DLL functions are invoked at runtime using these ordinal values. To illustrate this lets consider this scenario.

Example

Imagine that a DLL has to be developed and released as a 3rd part interface for some functionality. DLL, header and the corresponding import library. Now developers will link application with this import library. Now if an update had to be delivered or other interfaces have to be added , the dll is modified built and re-released. After modify the DLL, the function ordinals change, then that DLL is no longer compatible with the applications that use the older DLL, since the import library they link to Uses the previous function ordinals.

The import library and the DLL ordinal numbers must match. if applications linked with the older import library are run ? and on calling some import function the wrong function in the DLL will be invoked .whatever function is now corresponds to this ordinal will be invoked.

To illustrate this we have example Somedll.dll with functions

```

Class DllClass::Public CBase
{
IMPORT_C void SomedllFunction1();
IMPORT_C void SomedllFunction2();
};
?
..
..
..
EXPORT_C Class DllClass : void SomedllFunction1()
{
..
}
EXPORT_C Class DllClass : void SomedllFunction2()
{
..
}

```

Now the ordinals for the above exported function may look some thing like this

```

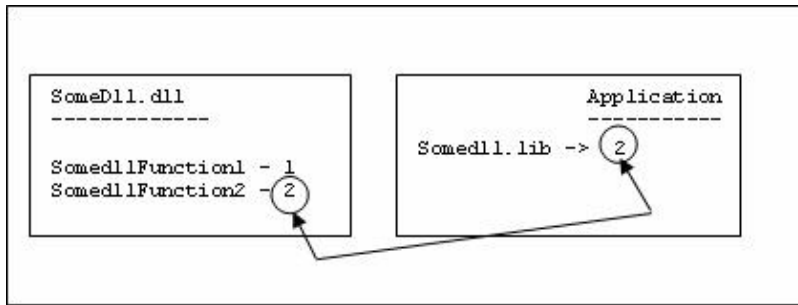
SomedllFunction1 ? 1
SomedllFunction2 - 2

```

After we release the dll along with the corresponding library the match is inline with the definition.

In client application calling the import library?s SomedllFunction2 will invoke the corresponding SomedllFunction2()function in the DLL by looking up function ordinal number 2.

Freeze



If some changes are required to be made to dll, the rebuild may result in `somedllFunction()` function getting a new ordinal number. Now the application developed using older dll and import library may not work because of this reason.