Building Symbian code for ARM Targets

Overview

Building for ARM will in general be more difficult than building for the emulator (WINS), and it is normal to find additional compiler errors and warnings from gcc on the first attempt. This is because Gcc is, in general, stricter than, say the Microsoft compiler, and also it has some subtle differences that will come out the first time an ARM build is attempted. The following covers a few of the most common pitfalls.

Function Exports

The gcc tool chain stricter than WINS builds when it comes to specified exported functions. The correct way to export a function from a DLL is as follows:

In the header(.h) file:

```
class CMyClass : public CBase
   {
IMPORT_C void Function();
   }
```

and then in the source(.cpp) file:

```
EXPORT_C void CMyClass::Function()
   {

   }
```

The WINS tool chain does not mind if the EXPORT_C is excluded from the CPP file; it exports the function anyway. However, the gcc tool chain requires the IMPORT_C and EXPORT_C to be perfectly matched. If they are not, the function will not be exported from the DLL, which will eventually lead to errors such as "Cannot Find Function" when attempting to link to this DLL.

NONSHARABLE_CLASS macro

The GCC compiler also differs from WINS compiler in that it always exports the class constructor, destructor and vtable. In classes that are meant to be instantiated and used outside the DLL this is desired, but not in internal classes. The difference can easily be checked by comparing the def files from emulator and ARM builds. The format of the lines in the file is irrelevant, but the amount of exports in the ARM def file is much greater because of the extra exports. The NONSHARABLE_CLASS macro has been defined for this reason. It is meant to be used when declaring a class that is not a part of the DLL exported API. It will tell the GCC compiler to not export the constructor, destructor and vtable.

```
NONSHARABLE_CLASS( CMyInternalClass ) : public CBase
    {
public:
    ...
```

Writable Static Data in DLLs

Other compilation differences include errors such as the "The MyDll.DLL has (un)initialized data" error arising from the build tool petran. The petran tool strips a PE format file (Win32 Portable Executable file format) of its irrelevant symbol information for an ARM target; thus making DLLs much smaller. As a consequence, ARM targets only support linking by ordinal. Petran is also responsible for adding UID information into the headers of executable files.

The Symbian OS architecture does not allow DLLs to have a data segment (static data, either initialized or uninitialized). There

(C) Copyright Nokia 2014. All rights reserved.

are fundamental problems in deciding whether a data segment should be used:

- Do all users of the DLL share it?
- Should it be copied for each process that attaches to the DLL?
- There are significant runtime overheads in implementing any of the possible answers.

However, as the WINS emulator uses the underlying Windows DLL mechanisms, it can provide per-process DLL data using "copy-on-write" semantics. This is why the problem goes undetected until the code is built for an ARM-based Symbian OS device.

Building and Freezing DLLs

When a DLL is loaded, it supplies a table of addresses, one for each exported symbol and one for the entry point of each exported function. This is the DLL's public interface, and DLLs should freeze their exports before release, so as to ensure the backward compatibility of new releases of a library. This is termed maintaining binary compatibility (BC), and the index of each export must remain constant from one release to another.

While you are developing a DLL, you can use the EXPORTUNFROZEN keyword in the .mmp file for the project to tell the build process that exports are not yet frozen. When you are ready to freeze, you must remove the EXPORTUNFROZEN keyword from the .mmp and supply a .def file listing the exports.

Symbian OS uses export definitions (.def) files to manage this requirement. Each exported symbol is listed in the exports section of the file with its ordinal number—ordinals start at 1.

The first time a build is done, a warning may be generated to say that the frozen .def file does not yet exist. Once the project has been completed, and has been built in its release form, you can freeze it with abld by using:

```
abld freeze
```

To maintain BC, every export defined in an earlier release must be defined in the new release. Any ordinals for new exports introduced in a new release must come after those defined in earlier releases.

For DLL builds, the command-line tools automatically create the .def file within the build tree for the specified target. Once these have been generated for a build, they can be archived with the project source and used in future builds to freeze the exports against change. This is done by copying the .def files into a default location and including the directive:

```
DEFFILE projectname.def
```

into the project .mmp file.

Normally only command-line builds should be released—note that some of the IDEs now support freezing of exports. In any subsequent command-line build of the project the exports will be guaranteed compatible with the current version.

If new exports are added, the new .def files should be copied from the build directory and archived with the new release.

(C) Copyright Nokia 2014. All rights reserved.

(C) Copyright Nokia 2014. All rights reserved.