

An Introduction to the Symbian OS Tool Chain

EMCC Software Ltd.

Published by the Symbian Developer Network

Version: 1.0 – January 2008

1 INTRODUCTION	2
2 AN OVERVIEW OF THE SYMBIAN OS TOOL CHAIN	2
2.1 The Standard C++ Compilation Process.....	2
2.2 Building Binaries for Symbian OS.....	3
2.3 A Look Behind the Scenes.....	6
2.4 Creating and Signing Installation Files.....	8
3 TOOLS IN MORE DETAIL	9
3.1 Devices	9
3.2 Makmake	10
3.3 The Symbian OS Resource Compiler.....	10
3.4 The Symbian OS Bitmap Converter.....	11
3.5 Post-Linker Tools: Elftran, Elf2e32 and Petran.....	11
3.6 Makedef	12
3.7 The Context-Sensitive Help Compiler.....	12
4 CUSTOM MAKEFILES	12
5 SUMMARY	13
6 BIOGRAPHY	14

1 Introduction

The Symbian OS tool chain is used to build applications for Symbian OS platforms. Most developers know the basic commands that are used to build binaries, but often they are not aware of what is happening in the background and which other tools are available. This white paper sheds some light on these often overlooked topics and aims to explain the Symbian OS tool chain in more detail.

All the tools described in this paper are command line tools. Integration with IDEs like CodeWarrior or Carbide.c++ is not covered.

Generally, the use of an IDE is recommended; however it is still useful to be aware of command line tools. Direct command line tool use, as opposed to indirect use through an IDE, is beneficial in the following ways:

- Provides an insight into what the IDE is doing “under the hood”.
- Allows direct intervention in the case of project or build corruption.
- Delivers improved efficiency in some cases; for instance, if only resources need to be regenerated for an ARMV5 release target, the command “`abl d resource armv5 urel`” will do just that and nothing else, whereas an IDE may not provide this level of control.
- Command line tools are scriptable, so the previous resource generation example could be turned into a script, enabling a non-technical user to produce customised versions of an executable – for example with customised images.
- Command line tool use allows the user to directly query the current environment settings and verify that the tools are running with, for instance, the correct PATH value by typing “`set path`”. This is important because some full-source SDKs require, and provide, specific versions of some tools, such as the RVCT compiler which need to be used in preference to the version currently installed on the PC. IDEs do not always provide a way to query all environment variables they’re using; Carbide.c++ does.

2 An Overview of the Symbian OS Tool Chain

For many developers the Symbian OS tool chain only seems to consist of two commands, namely `bl dmake` and `abl d`. These are the command line tools that you require when building binaries for Symbian OS. Even for many complex applications these commands alone are often sufficient.

But there are lots of other tools used in the background. This section first tries to explain the fundamentals of how to create binaries. It then outlines the usage of `bl dmake` and `abl d`, and finally it looks at other tools which are useful to know about.

2.1 The Standard C++ Compilation Process

Translating standard C++ source files into binary format is normally a well-understood process. It is done in two main steps: compiling and linking. Figure 1 outlines these steps:

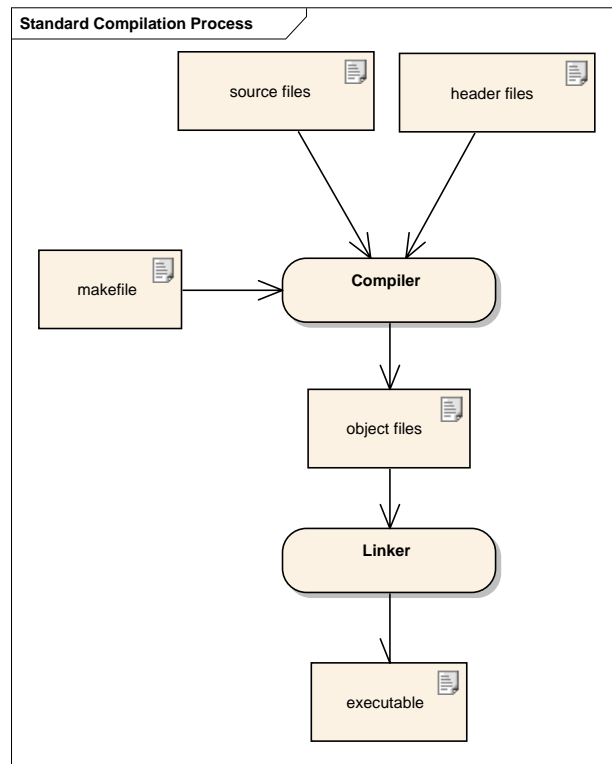


Figure 1: The Standard C++ Compilation Process

The compiler translates the source files into object files. Typically one object file per source file is created. Often a makefile (or any other compiler-specific project file) is used to specify which source files are part of the project. For very simple projects the source files can also be directly specified when calling the compiler.

The next step is to link these object files together. This is the responsibility of the linker, which takes all object files and creates one executable binary file. Depending on the type of binary, this can be an EXE or DLL file.

2.2 Building Binaries for Symbian OS

Having briefly explained the standard C++ compilation process, we can now look at how binaries are built specifically for Symbian OS. The basic principle is the same, although the tools from the previous section are not invoked directly, but by the Symbian tool chain.

2.2.1 Bldmake and Abld

The few simple steps required to create and build a Symbian OS project are as follows:

1. Create one or more project definition files (MMP files)
2. Create a component definition file (a **bld.inf** file)
3. Open a command prompt and run "bl dmake bl dfiles" in the directory in which **bld.inf** is located

4. Run “abl d bui l d”¹

The **bld.inf** file defines the components that form the project. It also allows the developer to specify any exported files, the platforms that are supported and any test components. For a detailed explanation of the syntax of the **bld.inf** file please refer to the Symbian OS SDK help² or open a command prompt and type “bl dmake i nf”.

The MMP file defines the specifics of one component. This includes the type of the component (for example whether the component is an executable, a DLL or a plugin), the list of source files that need to be built or any libraries this component links against. As for the **bld.inf** file, the syntax of the MMP file can be found in the Symbian OS SDK help.

The first command to call on every new project is bl dmake. It takes a **bld.inf** file as input and generates a number of makefiles (one per supported platform – see section 2.3 for details) and an **abl.d.bat** file. It is very easy to use; simply open a command prompt and type “bl dmake bl dfi les”.

Whereas bl dmake only needs to be called either at the beginning of a project or whenever the bld.inf file is modified, the next tool, abl d, is called whenever the source code needs to be compiled.

abl d accepts a number of parameters. To build the project for all supported platforms simply type “abl d bui l d” on the command prompt.

Figure 2 summarises the steps involved in building a project for Symbian OS:

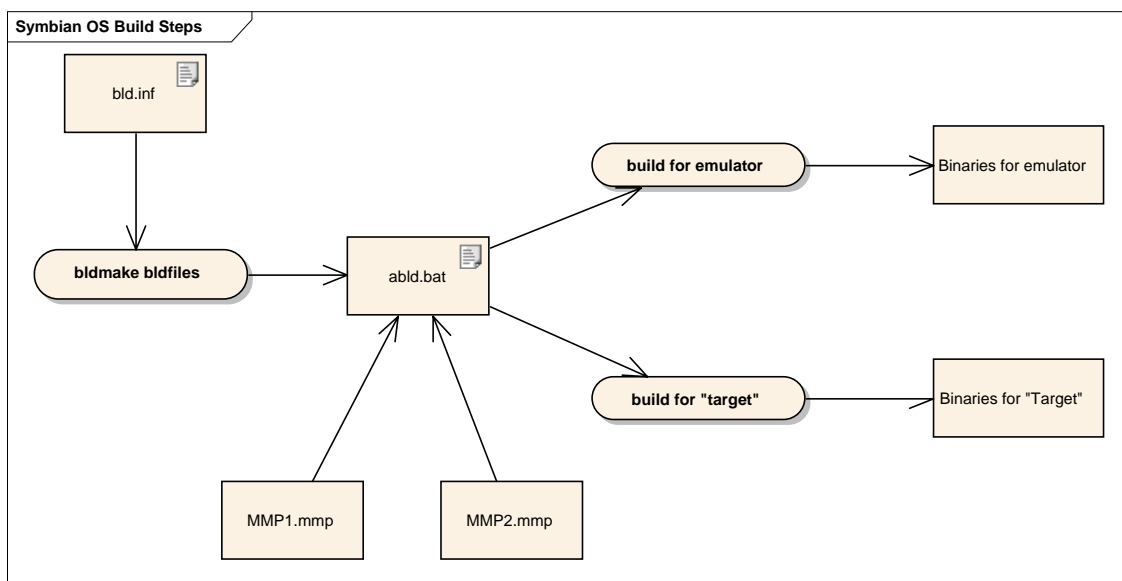


Figure 2: Symbian OS Build Steps

¹ Typically the platform and build target would be specified as well (e.g. abl d bui l d wi nscw udeb).

² Usually available within the documentation set supplied by a UI platform SDK (e.g. from UIQ or S60)

2.2.2 Target Types

When a component is translated into binary format, the tool chain needs to know what type of component needs to be built. This is done through the TARGETTYPE statement in the MMP file. Symbian OS supports a large number of target types, but the most common ones are:

- EXE – an executable program. This can be a GUI application, a server or a console application.
- DLL – a library that can be shared by multiple programs. This can either be a static interface DLL or a polymorphic interface DLL.
- PLUGIN – this indicates that the component is a plugin to the Symbian OS ECOM framework.

Please refer to the Symbian OS SDK help for information on other target types.

When building a static interface DLL, the tool chain creates a number of files: the DLL itself, an import library and a DEF file (unless EXPORTUNFROZEN is specified in the MMP file).

The DLL contains the library binaries and an exports table, which specifies the position of the exported functions via an ordinal number.

The import library (a LIB or DSO file – see section 2.2.3) specifies the exported functions of the DLL. It is used by the linker to resolve any function calls into the DLL. This involves a translation from function name to the ordinal number in the exports table. Import libraries must be included in the MMP file.

The DEF file is used to maintain binary compatibility between different versions of the DLL. As Symbian OS uses ordinal numbers rather than mangled function names for linking components it is important that the order of previously exported functions doesn't change. The DEF file species all exported functions and their associated ordinal number. When new exported functions are added, "abl d freeze" is used to re-generate the DEF file.

2.2.3 Compilers and Build Targets

Symbian OS supports a number of compilers and build targets. In the **bld.inf** file all build targets that the project supports can be specified. When calling "abl d bui l d", the project is built for all these supported platforms. Depending on the build target, the tool chain chooses different compilers to perform the build.

Three types of build targets are typically used. The first one is used when building for the emulator, and the command to call would be "abl d bui l d wi nscw udeb". The wi nscw argument specifies that the code should be built for the emulator on the Microsoft Windows platform, and udeb specifies that it is a debugging build. For this scenario, the Nokia x86 compiler, which comes with Carbide.c++, is used.

The other two build targets are used when building for a target device; the command to call depends on the compiler that is available. There are currently two compilers that can be used: the free GCC-E compiler that comes with the Symbian OS SDK or the commercial RVCT 2.2 compiler from ARM. The commands to run for a production build (indicated by urel) are "abl d bui l d gcc urel" for GCC-E and "abl d bui l d armv5 urel" for the RVCT compiler.

When building for ARMv5, the RVCT compiler produces binaries for the ARM v5 instruction set which, by default, comply with the application binary interface (ABI) version 1. The GCC-E compiler also builds for the ARM v5 instruction set, but only creates ABI version 2 binaries. ABI is a standard (created by ARM) that defines the specification a binary has to conform to. Binaries built

for ABI version 1 can interwork with binaries built for ABI version 2. More information on ARM ABIs can be found on the ARM website (www.arm.com).

When building a DLL, one important difference between these ABIs is that an ABI version 1 import library has the extension LIB, whereas a version 2 import library uses DSO as extension name. This should be considered when exporting a DLL interface – DSO files should be included with DLLs in an ABI version 2 release. Despite the extension change, and to ensure compatibility, the LIB extension can still be used in project MMP files for ABI v2 projects – DSO files will be correctly substituted in this case. DSO extensions specified in a project MMP file always force the use of DSO files, so an MMP file that specifies a DSO explicitly cannot be used in an ABI version 1 project.

Finally, it is also possible to create makefiles for either CodeWarrior or Visual Studio IDEs. This can be achieved by either running “`abl d makefi le cw_ide`” or “`abl d makefi le vs2003`” (Visual Studio 2005 or later is currently not officially supported). Please note that when using the vs2003 build target, Visual Studio uses the Nokia x86 compiler for building the project.

Carbide.c++ can import projects itself using a specified **bld.inf** file to locate associated MMPs. It does not require makefiles to be generated externally.

Figure 3 summarises the most common build targets:

BUILD TARGET	Description
WINSCW	Uses the Nokia x86 compiler to build the project for the emulator.
ARMv5	Uses the ARM RVCT 2.2 compiler to build for devices that support the ARMv5 instruction set (creates ABI v1 binaries).
ARMv5_ABIv2	Uses the ARM RVCT 2.2 compiler to build for devices that support the ARMv5 instruction set (creates ABI v2 binaries).
GCCE	Uses the GCC-E compiler to build for devices that support the ARMv5 instruction set (creates ABI v2 binaries).
ARMv6	Uses the ARM RVCT 2.2 compiler to build for devices that support the ARMv6 instruction set.
CW_IDE	Creates a project file for the CodeWarrior IDE.
VS6, VS2003	Creates a project file for the Visual Studio IDE (version 6 or 2003)

Figure 3: Common Build Targets

For all other supported build targets (like ARM4, ARMI or THUMB for versions prior to Symbian OS v9) please refer to the Symbian OS SDK help.

2.3 A Look Behind the Scenes

The two command line tools described in section 2.2 are wrappers around a much larger set of tools that are called in the background. Many of these tools are Perl based, which is why Perl needs to be installed in order to build applications for Symbian OS. This section sheds some light on these tools, and explains when and why they are used.

2.3.1 What Bldmake Does

As mentioned before, the `bldmake` command line tool processes the `bld.inf` file and, if that is successful, generates a number of files.

It creates a set of makefiles that are later used by `abld` to perform the build. These makefiles are typically referred to as first level makefiles. `Bldmake` creates two first level makefiles for each of the supported platforms: One for a normal build (with the name `<platform>.MAKE`) and one for building test code (with the name `<platform>.TEST.MAKE`). These makefiles are generated in `%EPOCROOT%\epoc32\build\<AbsolutePathToBld.InfFile>\`.

Additionally, `bldmake` creates the `abld` command line tool.

2.3.2 Abld and Makmake

`Abld` takes the first level makefiles and the project MMP files and performs the actual build. `Abld` is a very powerful tool, with many options. Luckily, most of its complexity is hidden, and many other tools are used in the background to perform a complete build.

`Abld` can be called with a number of commands. When a developer calls “`abld build`”, a combination of following commands are called in turn: `export`, `makefile`, `library`, `resource`, `target` and `final`.

1. “`abld export`” copies the files that are marked as exported (in the `bld.inf` file) to their destination.
2. “`abld makefile`” creates second level makefiles. In order to do that it uses the `make` program to process the first level makefile for the appropriate platform. The makefile label in the first level makefile invokes a tool called `makmake` for each MMP file. `Makmake` then creates a second level makefile specific for each MMP-file and platform. These second level makefiles are called `<MMP file name>.<platform>` and are located in `%EPOCROOT%\epoc32\build\<AbsolutePathToBld.InfFile>\<MMP file name>.\<platform>`.
3. “`abld library`” creates the import libraries for the DLLs in the project. It reads the DEF file and exports the appropriate functions in the order specified in the DEF file. A DEF file is used to maintain binary compatibility between builds. It defines the order of the exported functions.
4. “`abld resource`” builds resource and image files.

Resource files are built using the resource compiler. `epocrc.pl` is a Perl script that combines the pre-processing of the resource file and the compilation of the file, which is done by `rcomp`. This tool takes all resource files that are specified in the MMP file and generates compiled resource files (RSC) and resource file headers (RSG), if the `HEADER` label is specified in the MMP file for the particular resource.

Image files are processed using the bitmap converter `bmconv`. This tool packs all bitmap files that are listed in the MMP file into one multi-bitmap MBM file.

5. “`abld target`” builds the executables by invoking the appropriate compiler and linker for the specified platform. Additionally, the executables need to be converted into Symbian OS’s own executable format. This is done by either `elftran` or `elf2e32`. See section 3.5 for more details.
6. “`abld final`” enables extension makefiles to add final processing steps. Please refer to section 4 for more details.

In addition to these `abl d` commands there are certain commands a developer might want to call directly. These includes “`abl d clean`”, which cleans the executables, compiled resource files and MBM files that were created, and “`abl d real lycl ean`”, which additionally removes the files that were exported.

Another useful command is “`abl d bui l d -what`”. It shows exactly which files will be created by this build.

For other options on `abl d` please check out the Symbian OS SDK help.

2.4 Creating and Signing Installation Files

The previous sections have shown how to build an application for a particular platform. This build process produces a set of binary files which now need to be transferred to a device. However, before this can be done, another set of tools need to be called first. These tools are described in this section.

`Makesi s` is a command line tool that takes the binaries and packs them into an unsigned installation file (a SIS file). In order for the tool to know which binaries (and other files) need to be included, the developer has to provide a package file (PKG), which the tool expects as input. It is possible to specify a variety of options in the package file. Please refer to the Symbian OS SDK help for further information.

The general usage of this tool is “`makesi s <pkg fi l e name> <si s fi l e name>`”, but “`makesi s -h`” provides information on other available options.

Once a SIS file has been created, it can be transferred to a device. However, some applications need to be signed before they can be used, and some devices require all applications to be signed. Signing can either be performed by developers themselves or by submitting the application to a Symbian Signed test house. More information on this topic can be found at <http://www.symbiansigned.com>.

The `makekeys` tool encapsulates the creation of a public/private key pair and a certificate for self-signing. It can also be used to create certification requests for a certification authority.

The syntax of this tool can be displayed by calling `makekeys` without arguments.

Finally, `si gnsi s` signs the SIS file with the keys and certificate created by the previous tool (the certificate can also be a Developer Certificate from <http://www.symbiansigned.com>). The syntax for `si gnsi s` is “`si gnsi s <unsigned si s fi l e> <output> <certi fi cate fi l e> <key fi l e>`”. Further information can be found in the SDK help or by typing “`si gnsi s -?`” at the command line.

To simplify the usage of these tools, Symbian have created a wrapper tool called `CreateSi s`. If no keys or certificates are passes into this tool, it will automatically create a public/private key pair and a certificate for self-signing. Extensive help on this tool is available on the command line or in the SDK help.

Figure 4 shows all the steps involved in creating and signing a Symbian OS installation file:

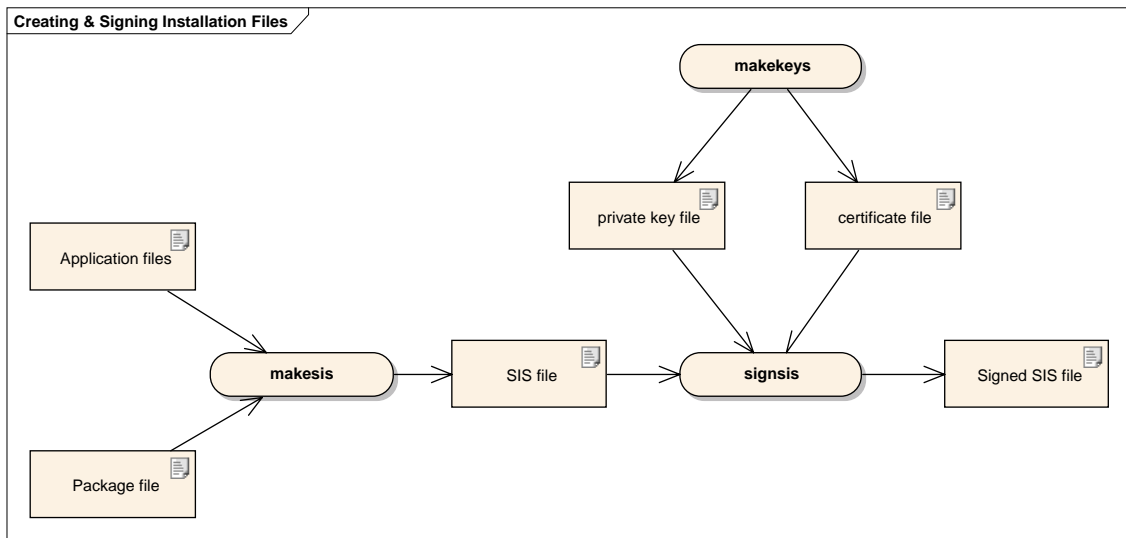


Figure 4: Creating and Signing a Symbian OS Installation File

3 Tools in More Detail

This chapter explains some of the tools mentioned in chapter 2 in more detail. It also discusses a couple of additional tools from the Symbian OS tool chain.

3.1 Devices

The `devices` command is useful when working with multiple SDKs on the same PC. It reads and writes to a file located at `C:\Program Files\Common Files\SYMBIAN\devices.xml`.

Generally, the `devices` command should be used to edit `devices.xml`, since manual editing runs the risk of file corruption. The purpose of the file is to select a particular SDK by overwriting the `EPOCROOT` environment variable temporarily. Each SDK tool has a tool stub in `C:\Program Files\Common Files\SYMBIAN` that reads the value from `devices.xml`, overwrites the `EPOCROOT` value, prepends a path to the `PATH` environment variable to point to the correct SDK's tools, and finally invokes the actual tool. `PATH` and `EPOCROOT` are reverted to their original values after the tool closes.

- To retrieve a list of all installed devices, use:
`devices`
- To query which is the current default device, use:
`devices -default t`
- To set the default device kit from the command line, use:
`devices --setdefault t @device-identifier`
- To override the current device kit in just the current command window, use:
`set EPOCDEVICE=device-identifier`
- The current device can be specified for individual commands:
`bl dmake bl dfiles @device-identifier`

Symbian OS version 7.0s and subsequent SDKs update `devi ces. xml` on installation, so it is rarely necessary to add an entry. For completeness, add an entry as follows:

```
devi ces -add c:\myepocroot c:\mytool sroot @devi ce. ID: com. Devi ce. Name
```

This produces:

```
<devi ce id="devi ce. ID" name="com. Devi ce. Name" default="no" userdel etabl e="yes">
<epocroot>c:\myepocroot</epocroot>
<tool sroot>c:\mytool sroot</tool sroot>
</devi ce>
```

in the `devi ces. xml` file.

Both the CodeWarrior and Carbide C++ IDEs are aware of device kits, and provide facilities for editing `devi ces. xml` and selecting the default device.

3.2 Makmake

Symbian OS allows source code to be built for multiple platforms. This includes building for the emulator or for various versions of hardware platforms. In order to support all these platforms, Symbian introduced a platform-neutral project file, called an MMP file. Information about the syntax of an MMP file can be found in the Symbian OS SDK help.

At some point during the build phase, this platform-neutral file needs to be converted into a makefile specific for a particular platform and compiler version. The tool that is responsible for this conversion is called `makmake`.

`Makmake` is typically called when the `MAKEFILE` label in the first level makefile is processed. It is called for each MMP file in the project, and its output is a second level makefile specific to one platform and one MMP file.

`Makmake` can also be called from the command line, though. You simply specify an MMP file (without extension) and a platform and you will get a second level makefile that can then be built using `make. exe`.

For example:

```
makmake test wi nscw
```

will build **test.mmp** for the Windows platform (i.e. the emulator) and produce a second level makefile called **test.wi nscw** in the same directory. This makefile can then be built by typing

```
make -f test. wi nscw
```

at the command prompt.

3.3 The Symbian OS Resource Compiler

Symbian OS development promotes a heavily resource-driven way of writing GUI applications. Most UI elements and strings are defined in resource files rather than C++ files. The format of a resource file allows for a very efficient compression of UI elements and text can be more easily localised if it resides in a separate resource file.

A Symbian OS resource file is a simple text file with a specific structure. Because text files are not optimal for low-memory devices, they first need to be translated into a more efficient binary format. This translation is the responsibility of the resource compiler.

Before a resource file can be converted, a C++ pre-processor processes the text file. This step includes replacing macros and copying the content of included files into the resource file. The conversion is then performed by a tool called `rcomp`.

These two steps are encapsulated by the `epocrc` tool. Not only is `epocrc` integrated into the `abl d` command, it can also be called directly from the command line. Typing just “`epocrc`” will display a list of possible options.

3.4 The Symbian OS Bitmap Converter

Most applications will provide one or more images or icons that need to be transferred to the device together with all other binaries. In order to save memory overhead, the Symbian OS tool chain packs all application images into one multi-bitmap file, called an MBM file. The list of images that goes into the MBM file is specified in the MMP file inside a START BITMAP / END section.

`bmconv` is the tool that performs the conversion into one file. Like the other tools mentioned so far, it is usually called in the background by the tool chain, but also can be called from the command line. This allows you to create MBM files without writing an MMP file first. By just typing “`bmconv`” the tool displays a list of possible options.

3.5 Post-Linker Tools: *Elftran*, *Elf2e32* and *Petran*

The post-linker tools were mentioned briefly in section 2; they are discussed here in more detail.

All three tools, `elftran`, `elf2e32` and `petran` have one thing in common: they translate the executables produced by the compiler tool chain into a native Symbian OS binary format which is called E32Image. The difference is explained in the following table:

Post-Linker Tool	Description
<code>elftran</code>	Translates binaries in ELF format (in ARM ABIv1) to the E32Image format.
<code>elf2e32</code>	Translates binaries in ELF format (in ARM ABIv2) to the E32Image format.
<code>petran</code>	Translates pre-Symbian OS v9 binaries to the E32Image format.

Figure 5: Post-Linker Tools

Symbian uses a proprietary file format for binaries to store extra information in those files as well as strip out unnecessary information. For example, platform security related information like capabilities is added, but mangled function names are removed (as they are called by ordinal – see sections 2.2.2 and 3.6).

The format of E32Image files changed in Symbian OS version 9. Previously, E32Image was based on the Portable Executable (PE) format that is used for Microsoft Windows binaries. Since Symbian OS version 9, E32Image is based on the UNIX-style Executable and Linking format (ELF).

All these tools can be extremely useful for developers as they not only do the transformation between the two file formats, but also provide information about binaries. By simply typing the command, the tools provide a list of options that allow you to retrieve and modify information like the UIDs of the binary, its capabilities, and much more.

For example, it is possible to list the capabilities of a binary by typing

```
elftran -dump s Application.exe
```

and then modify them by calling

```
el ftran -capability "NetworkServices AllFiles" Application.exe
```

If you were to call the first command again you will see that the application now has NetworkServices and AllFiles capabilities.

3.6 Makedef

In order to use the often limited memory in mobile devices most efficiently, function names in Symbian OS DLLs are not resolved via mangled names but via ordinals. This can potentially cause binary compatibility issues, if not done carefully. To support binary compatibility, Symbian OS uses DEF files, which specify the order of the exported functions of a DLL.

This DEF file is created by a tool called makedef. It is invoked when calling “abl d freeze” from the command prompt. Makedef creates a new DEF file if one doesn’t exist, or otherwise it compares the content of the DEF file with the current list of exported functions. If the current list is a superset of that in the DEF file, makedef adds the new functions to the end of the list in the DEF file. Otherwise it reports an error.

3.7 The Context-Sensitive Help Compiler

The Symbian OS context-sensitive help compiler cshl pcmp is a command line tool that supports the creation of context-sensitive help files for applications. The Symbian OS SDK help provides extensive documentation for this tool, and an example project, so this section is only a brief overview.

The source for all context-sensitive help content is stored in RTF files. In order to specify which RTF files should be part of the project, a project file needs to be created. This is a simple XML file with the extension CSHLP. Optionally, a customisation file can be created to change the appearance of the help content.

When all these files are written, cshl pcmp is called in the command line to build the help file. The syntax is very simple: “cshl pcmp <project-file.cshlp>”. This help file can then be verified in the Symbian OS emulator to confirm its accuracy. Note that the supported features may vary between UI platform.

4 Custom Makefiles

The Symbian OS tool chain provides the possibility to extend the default build process by allowing additional makefiles to be plugged-in. Such makefiles can be useful for automating tasks that should be performed whenever the project is built.

Custom makefiles look like any other makefiles, but in order to integrate them into the Symbian OS tool chain they have to handle certain make targets:

Make Target	Called by following ABLD command
makmake	makefile
freeze	freeze
lib	library

cleanlib	tidy
clean	clean
final	final
resource	resource
bld	target
savespace	target –savespace
releasables	target [-what -check]

Figure 6: Mandatory Make Targets for Custom Makefiles

The “final” make target can be used when any tasks performed in the custom makefile rely on the fact that the build has been completed.

When performing a build, `abl d` will then call the appropriate make target for all included custom makefiles (in addition to the ones in the makefiles that were created by the Symbian OS tool chain). It is possible to leave these make targets blank (i.e. no commands are called), but the labels have to exist.

Here is an example from the Symbian OS SDK help, where only the `makmake` target contains a command (in this case only an `echo` command). All other targets are left blank.

```
MAKMAKE :
    echo this is an example
FINAL FREEZE LIB CLEANLIB RESOURCE RELEASABLES CLEAN BLD SAVESPACE :
```

In order to include a custom makefile in a project, it must be listed in the `bld.inf` file. This is done by using one of these three statements in the `PRJ_MMPFILES` section: `makefile`, `nmakefile` or `gnumakefile`. The first two variants use the makefile with Microsoft’s `nmake.exe`, whereas `gnumakefile` uses it with `make.exe`.

5 Summary

This white paper discussed the various tools of the Symbian OS tool chain. It explained tools like `bl dmake` and `abl d`, which are used by developers on a regular basis, but also covered tools that are typically only called in the background.

Even if it is not absolutely necessary to understand the entire tool chain in detail, it is often useful to have at least a basic understanding of it. This allows developers to make modifications to it (for example through custom makefiles), and can help understanding warnings and error messages better.

Most of the command line tools are fairly powerful and the intention of this document was not to duplicate the information that is already easily available in the Symbian OS SDK help or as a command line help. Please refer to these sources if more information on any of the Symbian OS tools is required.

6 Biography



EMCC Software is a leading provider of mobile software solutions and consultancy services. As a renowned Symbian, S60 and UIQ Competence Centre focusing on mobile and associated technologies, EMCC has access to OS and platform roadmaps and full source code which greatly expands our ability to solve

complex development issues. EMCC has become a preferred supplier to many platform providers and handset manufacturers.

EMCC's core competencies include development of solutions using technologies such as VoIP, Wi-Fi, SIP based functionality, messaging and communications for a variety of platforms, handsets and enterprise solution providers. Further details are available from <http://www.emccsoft.com/technology>

Want to be kept informed of new articles being made available on the Symbian Developer Network?

[Subscribe to the Symbian Community Newsletter.](#)

The Symbian Community Newsletter brings you, every month, the latest news and resources for Symbian OS.