

1

Getting Started

It seems to be traditional to start a book on computer programming with a 'Hello World' example and, although this book is more about an operating system than a programming language, we are following that tradition. In the process we introduce you to the emulator and to the tools for building C++ programs, so that by the end of the chapter you will know how to build and run a Symbian OS application. We don't get too involved in describing Symbian OS programming conventions, API functions, and so forth; instead, we concentrate on the tools you need and how to use them, leaving the details until later chapters.

First we briefly describe the emulator. Most Symbian OS software is developed first on the emulator and only then on real target hardware. The emulator also includes a number of Symbian OS applications, and so mimics a real Symbian OS phone very closely. You will need to become familiar with the emulator and in the process we can take a look at the various graphical user interfaces (GUIs) used by Symbian OS.

Then we create a program. The easiest things to build are text-mode console programs, so that's the form of the classic 'Hello World' application that we use. We demonstrate how to compile it for the emulator, and how to launch it using the Carbide.c++ IDE.

1.1 Using the Emulator

The emulator is a fundamental tool for all the Symbian OS SDKs, so it's vital that you get to know how to use it.

If you are a newcomer to Symbian OS, the emulator offers an opportunity to get to know some Symbian OS basics from a user's perspective, so we look at these straight away. Later, you'll want to learn to make

effective use of it as a developer, so we cover the details of its operation in Chapter 10.

If you have some experience of Symbian OS, you may want to skip straight to Section 1.2 and start building an application.

Launching the Emulator

The first piece of software you need is a software development kit (SDK). There are a couple of Symbian OS v9 SDKs available, depending on the phone(s) you want to target. If you're unsure which SDK to select, we recommend starting with both a S60 v3 and a UIQ 3rd Edition SDK. You can obtain these via the links on the Symbian developer website (developer.symbian.com). Once you've installed your SDK, you can launch the emulator in any of the following ways:

- launch the executable `epoc.exe`, which you'll find in the directory `\epoc32\release\winscw\udeb` underneath the directory in which the SDK is installed
- from the Start menu, select either Programs, UIQ SDK or Programs, S60 Developer Tools, 3rd Edition SDK and select Emulator from the appropriate submenu.

However you choose to start it, and whichever emulator you're using (either for S60 or UIQ), the first thing you'll see in the emulator is the application launcher. As its name indicates, the application launcher enables you to launch applications. Its menus allow you to view or change system settings and it also has a control panel. It's very easy for end users to get to know the application launcher; you don't really need a manual. Just click with the mouse here and there and you'll soon find out what it has to offer.

GUI Style

If you've started up the UIQ emulator (shown in Figure 1.1a with the P990 extensions), as you browse around the application launcher you'll begin to see how UIQ is optimized for the pen-based mobile phone form factor. UIQ is designed as a 'read mostly' user interface, to be used mainly for browsing and for making a selection from a range of options with a single tap of a pen. Other GUIs – such as the S60 interface shown in Figure 1.1b and used, for example, on the Nokia Nseries phones – are optimized for the different hardware resources of the devices on which they are intended to run.

Although the various GUIs may have a superficially different appearance, they all rely on a common set of underlying features, some of which are briefly described in the next section.

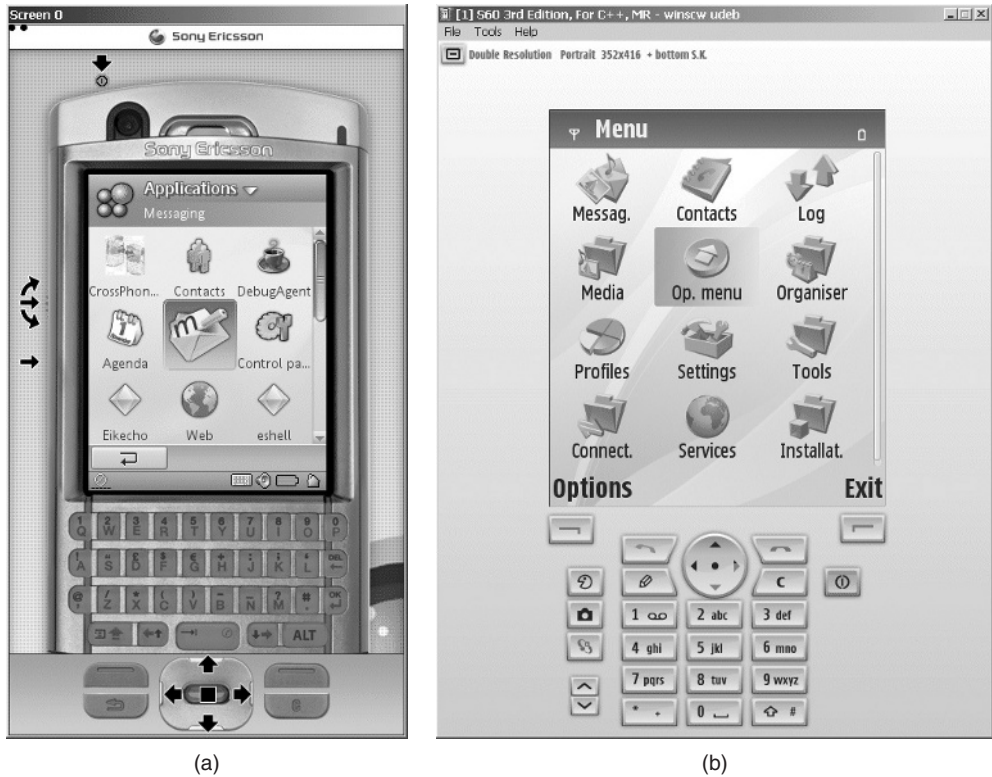


Figure 1.1 UIQ (with P990 extensions) and S60 emulators

Screen layout

The UIQ screen layout, illustrated in the following diagram, includes the following areas (from top to bottom of the screen):

The *title bar* displays the name and icon of the current application. It also contains a *View context area*. Selecting the label opens the menu in Pen style. The width of the menu is adjusted to fit the longest label. The *View context area* can contain controls, for example icons, text labels and tabs. If there is enough space to the right of the tabs, other application-specific data or controls can be added.

The *menu bar* contains one or more menus, whose names and contents change from application to application, and also as you change view within a particular application. In UIQ, the menu bar usually contains two menus on the left and may optionally have a folder menu on the right.

The *application space* is the central area of the screen, where an application's view is displayed. Applications use this area in whatever way is appropriate to the information that they display.

Optionally, an application displays a *button bar* at the bottom of the application space. The most common use is to provide buttons to move

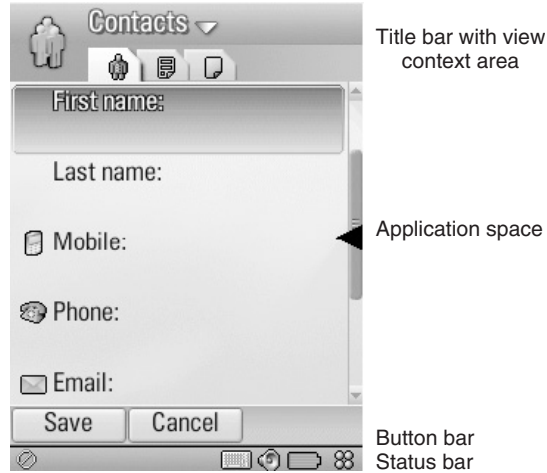


Figure 1.2 UIQ screen layout

between the application's various views. In UIQ, a detail view, such as the one in Figure 1.2 that shows the detail of a single Contact entry, usually has a special button in the lower right corner to return you to the main view.

The *status bar* displays information such as battery charge, time of day, signal strength and notification of incoming messages. The P990i's status bar includes a keyboard icon in the lower right corner, which is used to display a virtual keyboard for text input if you do not wish to use handwriting recognition.

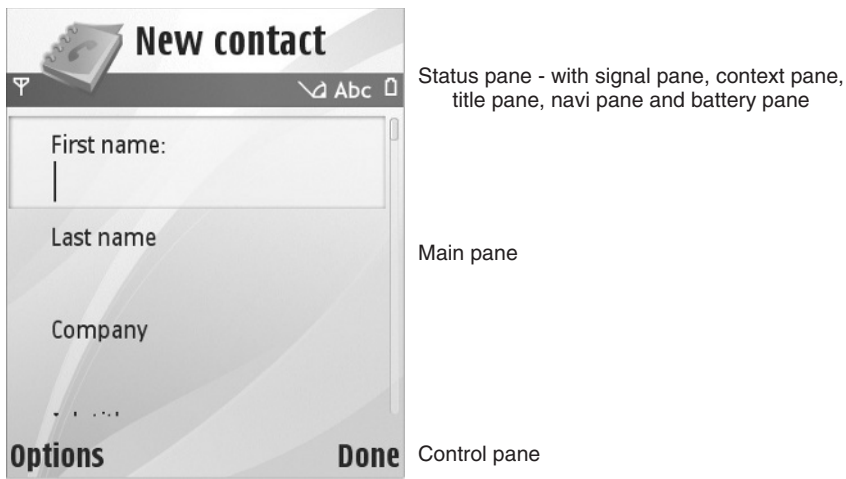


Figure 1.3 S60 screen layout

Also displayed in some views is the *application picker*, containing icons that allow you to switch applications. Selecting an icon brings the application it represents to the foreground. The application launcher icon brings the application launcher to the foreground, allowing you to launch applications that are not displayed on the application picker. If you wish, you can customize the application picker to launch your own preferred set of applications.

Most of these screen layout elements can be recognized in other GUIs used with Symbian OS, such as in Figure 1.3, though they may differ significantly in appearance or be located in different areas of the screen.

Menus

Figure 1.4 illustrates a set of menus in the Calendar application. The menu bar is different (but not very different) from menu bars in desktop GUIs.

Cascaded menu items can be used both to hide less common options and to reduce the vertical space required by menu panes. This feature is used sparingly in UIQ applications, where menu content changes with context and each menu is designed to contain as small and as simple a set of menu commands as possible. Cascaded menu items are used more frequently in the S60 interface, where they appear as in Figure 1.5.

In the UIQ emulator, use the menu with the pen (i.e. your PC's mouse) and you'll see some interesting visual cues to confirm the option you



Figure 1.4 Menu structure in the Calendar application



Figure 1.5 S60 cascaded menu items

selected: the option will flash very briefly before the menu disappears. It took a long time to get that effect just right!

As with all other elements of a Symbian OS GUI and applications, where a keyboard is available (including the keyboard of a PC running the emulator), you can drive the menus with the keyboard as well as with the pen. You can use the arrow keys and Enter to select items. You can also use cursor keys and the Confirm button on real target hardware and on emulators that support such features.

When writing an application you also have the option to assign a shortcut key to any menu item, which allows you to invoke the relevant function directly from a keyboard without going through the menus at all. Although they can be defined in any Symbian OS application, shortcut keys are clearly not usable on mobile phones without keyboards (except when the application is running on the emulator) so neither the UIQ nor the S60 user interfaces display shortcut key information in their menus.

1.2 Hello World – Text Version

Now that you've started to get to grips with the emulator, it's time to get your first Symbian OS C++ program running. Even though Symbian OS is primarily a system for developing GUI applications, the simplest kind of program uses a text interface, so for our first task we'll build a program

that writes ‘Hello world!’ to a text console. That will introduce you to the tools required for building applications for both the emulator and a real device, so that later on you’ll be ready for a program with a GUI.

If you want to follow this chapter through at your desktop with the SDK, make sure that you’ve installed all the tools you need. See the appendix for more information.

The Program: **HelloText**

Here’s the program we’re going to build. It’s your first example of Symbian OS C++ source code:

```
// helloworld.cpp

#include <e32base.h>
#include <e32cons.h>

LOCAL_D CConsoleBase* gConsole;

// Real main function
void MainL()
{
    gConsole->Printf(_L("Hello world!\n"));
}

// Console harness
void ConsoleMainL()
{
    // Get a console
    gConsole = Console::NewL(_L("Hello Text"),
                            TSize(KConsFullScreen, KConsFullScreen));
    CleanupStack::PushL(gConsole);

    // Call function
    MainL();

    // Pause before terminating
    User::After(5000000); // 5 second delay

    // Finished with console
    CleanupStack::PopAndDestroy(gConsole);
}

// Cleanup stack harness
GLDEF_C TInt E32Main()
{
    __UHEAP_MARK;
    CTrapCleanup* cleanupStack = CTrapCleanup::New();
    TRAPD(error, ConsoleMainL());
    __ASSERT_ALWAYS(!error, User::Panic(_L("Hello world panic"), error));
    delete cleanupStack;
    __UHEAP_MARKEND;
    return 0;
}
```

Our main purpose here is to understand the Symbian OS tool chain, but while we have the opportunity, there are a few things to observe in the source code above. There are three functions:

- the actual ‘Hello world!’ work is done in `MainL()`
- `ConsoleMainL()` allocates a console and calls `MainL()`
- `E32Main()` allocates a trap harness and then calls `ConsoleMainL()`.

On first sight, this looks odd. Why have three functions to do what most programming systems can achieve in a single line? The answer is simple: real programs, even small ones, aren’t one-liners. So there’s no point in optimizing the system design to deliver a short, sub-minimal program. Instead, Symbian OS is optimized to meet the concerns of real-world programs – in particular, to handle and recover from memory allocation failures with minimal programming overhead. There’s a second reason why this example is longer than you might expect: real programs on a user-friendly machine use a GUI framework rather than a raw console environment. If we want a console, we have to construct it ourselves, along with the error-handling framework that the GUI would have included for us.

Error handling is of fundamental importance in a machine with limited memory and disk resources, such as those for which Symbian OS was designed. Errors are going to happen and you can’t afford not to handle them properly. We’ll explain the error-handling framework and its terminology, such as trap harness, cleanup stack, leave and heap marking, in Chapter 4.

The Symbian OS error-handling framework is easy to use, so the overheads for the programmer are minimal. You might doubt that, judging by this example! After you’ve seen more realistic examples, you’ll have better grounds for making a proper judgement.

Back to `HelloText`. The real work is done in `MainL()`:

```
// Real main function
void MainL()
{
    gConsole->Printf(_L("Hello world!\n"));
}
```

The `printf()` that you would expect to find in a C ‘Hello World’ program has become `Console>Printf()` here. That’s because Symbian OS is object-oriented: `Printf()` is a member of the `CConsoleBase` class.

The `_L` macro turns a C-style string into a Symbian OS-style descriptor. We’ll find out more about descriptors, and a better alternative to the `_L` macro, in Chapter 5.

Symbian OS always starts text programs with the `E32Main()` function. `E32Main()` and `ConsoleMainL()` build two pieces of infrastructure needed by `MainL()`: a cleanup stack and a console. Our code for `E32Main()` is:

```
// Cleanup stack harness
GLDEF_C TInt E32Main()
{
    __UHEAP_MARK;
    CTrapCleanup* cleanupStack = CTrapCleanup::New();
    TRAPD(error, ConsoleMainL());
    __ASSERT_ALWAYS(!error, User::Panic(_L("Hello world panic"), error));
    delete cleanupStack;
    __UHEAP_MARKEND;
    return 0;
}
```

The declaration of `E32Main()` indicates that it is a global function. The `GLDEF_C` macro, which in practice is only used in this context, is defined as an empty macro in `e32def.h`. By marking a function `GLDEF_C`, you show that you have thought about it being exported from the object module. `E32Main()` returns a `TInt` integer. We could have used `int` instead of `TInt`, but since C++ compilers don't guarantee that `int` is a 32-bit signed integer, Symbian OS uses `typedefs` for standard types to guarantee they are the same across all Symbian OS implementations and compilers.

`E32Main()` sets up the error-handling framework. It sets up a cleanup stack and then calls `ConsoleMainL()` under a trap harness. The trap harness catches errors – more precisely, it catches any functions that leave. If you're familiar with exception handling in standard C++ or Java, `TRAP()` is like `try` and `catch` all in one, `User::Leave()` is like `throw`, and a function with `L` at the end of its name is like a function with `throws` in its prototype.

Here's `ConsoleMainL()`:

```
// Console harness
void ConsoleMainL()
{
    // Get a console
    gConsole = Console::NewL(_L("Hello Text"),
        TSize(KConsFullScreen, KConsFullScreen));
    CleanupStack::PushL(gConsole);

    // Call function
    MainL();

    // Pause before terminating
    User::After(15000000); // 15 second delay

    // Finished with console
    CleanupStack::PopAndDestroy(gConsole);
}
```

This function allocates a console before calling `MainL()` to do the `Printf()` of the 'Hello world!' message. After that, it briefly pauses and then deletes the console again.

If we were creating this example for a target machine that had a keyboard, we could have replaced the delay code:

```
// Pause before terminating
User::After(15000000); // 15 second delay
```

with something like:

```
// Wait for key
console->Printf(_L("[ press any key ]"));
console->Getch(); // Get and ignore character
```

so that the application would wait for a keypress before terminating.

There is no need to trap the call to `MainL()`, because a leave would be handled by the `TRAP()` in `E32Main()`.

The main purpose of the cleanup stack is to prevent memory leaks when a leave occurs. It does this by popping and destroying any object that has been pushed to it. So if `MainL()` leaves, the cleanup stack will ensure that the console is popped and destroyed. If `MainL()` doesn't leave, then the final statement in `ConsoleMainL()` will pop and destroy it anyway.

In fact, in this particular example `MainL()` cannot leave, so the `L` isn't theoretically necessary. But this example is intended to be a starting point for other console-mode programs, including programs that do leave. We've left the `L` there to remind you that it's acceptable for such programs to leave if necessary.

If you're curious, you can browse the headers: `e32base.h` contains some basic classes used in most Symbian OS programs, while `e32cons.h` is used for a console interface and therefore for text-mode programs – it wouldn't be necessary for GUI programs. You can find these headers (along with the headers for all Symbian OS APIs) in `\epoc32\include` on your SDK installation drive.

The Project Specification File

As in all C++ development under Symbian OS, we start by building the project to run under the emulator (that is, for an x86 instruction set) using the Carbide.c++ compiler. We use a debug build so that we can see the symbolic debug information and to get access to some useful memory-leak checking tools. In Chapter 9 we'll build the project for a target Symbian OS phone, using an ARM instruction set. At that stage we'll use the release build, since that is what you would eventually do to create your final, usable, application.

For demonstration purposes we're actually going to build the project twice, because you can either compile the code from the command line or build it in the Carbide.c++ IDE.

Each type of build requires a different project file. To simplify matters, you put all the required information into a single generic *project specification file*, and then use the supplied tools to translate that file into the makefiles or project files for one or more of the possible build environments. Project specification files have a `.mmp` extension (which stands for 'makmake project'). The one for the `HelloText` project is as follows:

```
// helloworld.mmp
TARGET      HelloText.exe
TARGETTYPE  exe
SOURCEPATH  .
SOURCE      helloworld.cpp
USERINCLUDE .
SYSTEMINCLUDE \epoc32\include
LIBRARY     euser.lib
```

This is enough information to specify the entire project, enabling configuration files to be created for any platform or environment.

- The `TARGET` specifies the executable to be generated and the `TARGETTYPE` confirms that it is an EXE.
- `SOURCEPATH` specifies the location of the source files for this project.
- `SOURCE` specifies the single source file, `helloworld.cpp` (in later projects, we'll see that `SOURCE` can be used to specify multiple source files).
- `USERINCLUDE` and `SYSTEMINCLUDE` specify the directories to be searched for user include files (those included with quotes; the `SYSTEMINCLUDE` path is searched as well) and system include files (those included with angle brackets; only the `SYSTEMINCLUDE` path is searched). All Symbian OS projects should specify `\epoc32\include` for their `SYSTEMINCLUDE` path.
- `LIBRARY` specifies libraries to link to – these are the `.lib` files corresponding to the shared library DLLs whose functions you will be calling at run time. In the case of this very simple program, all we need is the E32 user library, `euser.lib`.

The Component Definition File

The Symbian OS build tools require one further file, the *component definition file*, to be present. This file always has the name `blt.inf` and contains a list of all the project definition files (frequently, there is only one) that make up the component. In more complex cases it will usually

contain further build-related information, but the one for `HelloText` is simply:

```
// BLD.INF
PRJ_MMPFILES
hellotext.mmp
```

Building from the Command Line

Once you've typed in the example code for the three files as listed above, we can begin to compile the application.

To start the command-line build, open up a command prompt, change to your installation drive and go to the source directory containing the code for this example. The first stage is to invoke `bldmake` by typing:

```
bldmake bldfiles
```

After a short pause, this command will return. By default, `bldmake` doesn't tell you anything. However, if you check the contents of the directory, you'll notice a new file, `abld.bat`, that is used to drive the remainder of the build process. You will also find that there is a new directory in the `\epoc32\build` directory tree, containing a number of generated files which relate to the various types of build that the build tools support.

Next, use `abld` to run the rest of the build by typing:

```
abld build winscw udeb
```

The `winscw` parameter specifies that we are building for the emulator, using the Carbide.c++ compiler, and the `udeb` parameter means we are using a (unicode) debug build. The command generates the following output:

```
make -r -f "\EPOC32\BUILD\HELLOTEXT\EXPORT.make" EXPORT VERBOSE=-s
Nothing to do
make -r -f "\EPOC32\BUILD\HELLOTEXT\WINSCW.make" MAKEFILE VERBOSE=-s
perl -S makmake.pl -D \HELLOTEXT\HELLOTEXT WINSCW
make -r -f "\EPOC32\BUILD\HELLOTEXT\WINSCW.make" LIBRARY VERBOSE=-s
make -s -r -f "\EPOC32\BUILD\HELLOTEXT\HELLOTEXT\WINSCW\
HELLOTEXT.WINSCW" LIBRARY
make -r -f "\EPOC32\BUILD\HELLOTEXT\WINSCW.make" RESOURCE
CFG=UDEB VERBOSE=-s
make -s -r -f "\EPOC32\BUILD\HELLOTEXT\HELLOTEXT\WINSCW\
HELLOTEXT.WINSCW" RESOURCEUDEB
make -r -f "\EPOC32\BUILD\HELLOTEXT\WINSCW.make" TARGET CFG=UDEB
VERBOSE=-s
make -s -r -f "\EPOC32\BUILD\HELLOTEXT\HELLOTEXT\WINSCW\
HELLOTEXT.WINSCW" UDEB
make -r -f "\EPOC32\BUILD\HELLOTEXT\WINSCW.make" FINAL CFG=UDEB VERBOSE=-s
```

The build is split into six phases:

- The export phase copies exported files to their destinations. This typically includes copying public header files into the `\epoc32\include` directory. For many applications, as in the current case, this stage needs to do nothing.
- The makefile phase creates the necessary makefiles or IDE workspaces.
- The library phase creates import libraries.
- The resource phase creates the application's resources files, bitmaps and information files.
- The target phase creates the application's main executables.
- The final phase is present to perform any final actions that need to be done after the main executables have been created. For most applications, this phase does nothing.

These phases, and other possible options available with the `abld` tool, are fully described in the Build Tools Guide and Build Tools Reference sections of the Developer Library documentation supplied with Symbian OS SDKs. Typing `abld help` also gives a useful summary of the options available.

The result of running the `abld` tool is that the `HelloText` project is built into the emulator startup directory as `\epoc32\release\winscw`



Figure 1.6 HelloText emulator output

`\udeb\hellotext.exe`. To run the program under the UIQ emulator, you can start it right from there, using either the command prompt or Windows Explorer. The emulator will boot and you'll see 'Hello world!' on the screen for a few seconds, as illustrated in Figure 1.6.

On the S60 emulator, the program can be run in the same way, but is hidden behind the application picker display. To bring it to the foreground, once the main display is showing, press the Applications key for a couple of seconds to see the task list. Then (as `HelloText` is the only application you're running) press the Select soft key to show `HelloText` running. But be quick – you've only got 15 seconds!

Using Multiple SDKs

If you've installed more than one SDK, you need to ensure that the tools used when you build an application belong to the SDK that you're currently using. To help with this, there is a `devices` command in Symbian OS SDKs based on v7.0 and later. To list the SDKs which you currently have installed, just type `devices` at the command prompt. The SDK currently in use has 'default' appended to the name. To select a different SDK, type `devices -setdefault @<sdk>`, where `<sdk>` is the full name as copied from the original `devices` list.

Building in the Carbide.c++ IDE

Now we know our tool chain is working, let's build the project from the IDE and use the debugging tools on the example. First, start up the Carbide.c++ IDE and select File, Import and choose the option Symbian MMP File.

Browse to the directory containing the source code and select `hellotext.mmp`. Select the 'Emulator debug' build configuration for the appropriate SDK and then click on Finish; after a short time, the project will have been created.

You can build the project straight away by selecting Project, Build Project – the default target is `WINSCW UDEB`, as we used when building from the command line.

Once the code has been compiled, you can launch the emulator from the IDE by selecting Run, Run HelloText... Don't forget that you'll need to bring `HelloText` to the foreground if you're using the S60 emulator. Alternatively, you can run the debugger on the code by selecting Run, Debug HelloText... You can then use any of the usual debug techniques – run to cursor, step over a whole line of code, step into each of the functions on a line, step out of the current function, run to breakpoint, etc.

If you're curious, you might want to try debugging through line-by-line. You'll begin to get a feel for what's worth doing and what's not, and it will give you some insight into the system structure. On the other hand,

there's no need to jump in this deep right now. We'll explain what you really need to know through the next few chapters. The main point to note is that the Carbide.c++ IDE provides an excellent debugger, and as a Symbian OS developer you can take full advantage of it. We'll look at debugging in more detail in Chapter 10.

Summary

In this chapter, we've not gone very heavily into code, but have instead focused on the tools that come with the SDK and how to use them to build and test a simple project.

The topics we've looked at are:

- how to use the emulator
- a hint as to the support Symbian OS offers to help you code safely
- the basic structure of the project specification (MMP) file
- using the Carbide.c++ IDE and command-line tools
- building and running applications on the emulator.

