Rapid Mobile Enterprise Development for Symbian OS

An Introduction to OPL Application Design and Programming

Ewan Spence

With
Phil Spencer and Rick Andrews

Reviewed by
Phil Spencer

Managing editor
Phil Northam

Assistant editor
William Carnegie

John Wiley & Sons, Ltd

Rapid Mobile Enterprise Development for Symbian OS

TITLES PUBLISHED BY SYMBIAN PRESS

- Wireless Java for Symbian Devices
Jonathan Allin
0471 486841 512pp 2001 Paperback

- Symbian OS Communications Programming
Michael J Jipping
0470 844302 418pp 2002 Paperback

- Programming for the Series 60 Platform and Symbian OS
Digia
0470 849487 550pp 2002 Paperback

- Symbian OS C++ for Mobile Phones, Volume 1
Richard Harrison
0470 856114 826pp 2003 Paperback

- Programming Java 2 Micro Edition on Symbian OS
Martin de Jode
0470 092238 498pp 2004 Paperback

- Symbian OS C++ for Mobile Phones, Volume 2
Richard Harrison
0470 871083 448pp 2004 Paperback

- Symbian OS Explained
Jo Stichbury
0470 021306 448pp 2004 Paperback

- PC Connectivity Applications for Symbian OS
Ian McDowall
0470 090537 480pp 2004 Paperback

Rapid Mobile Enterprise Development for Symbian OS

An Introduction to OPL Application Design and Programming

Ewan Spence

With
Phil Spencer and Rick Andrews

Reviewed by
Phil Spencer

Managing editor
Phil Northam

Assistant editor
William Carnegie

John Wiley & Sons, Ltd

Other Wiley Editorial Offices

John Wiley & Sons Inc., 111 River Street, Hoboken, NJ 07030, USA

Jossey-Bass, 989 Market Street, San Francisco, CA 94103-1741, USA

Wiley-VCH Verlag GmbH, Boschstr. 12, D-69469 Weinheim, Germany

John Wiley & Sons Australia Ltd, 33 Park Road, Milton, Queensland 4064, Australia

John Wiley & Sons (Asia) Pte Ltd, 2 Clementi Loop #02-01, Jin Xing Distripark, Singapore 129809

John Wiley & Sons Canada Ltd, 22 Worcester Road, Etobicoke, Ontario,
Canada M9W 1L1

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Library of Congress Cataloging-in-Publication Data

Spence, Ewan.
Rapid mobile enterprise development for Symbian OS : an introduction to OPL application design and programming / Ewan Spence, with Phil Spencer and Rick Andrews.
p. cm.
Includes bibliographical references.
ISBN-13 978-0-470-01485-1 (alk. paper)
ISBN-10 0-470-01485-7 (alk. paper)
1. Cellular telephone systems–Computer programs. 2. Operating systems (Computers) 3. OPL (Computer program language) I. Spencer, Phil. II. Andrews, Rick. III. Title.
TK6570.M6S66 2005
005.26′8 – dc22

2004027113

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library

ISBN-13 978-0-470-01485-1
ISBN-10 0-470-01485-7

Typeset in 10/12pt Optima by Laserwords Private Limited, Chennai, India
Printed and bound in Great Britain by Biddles Ltd, King's Lynn
This book is printed on acid-free paper responsibly manufactured from sustainable forestry in which at least two trees are planted for each one used for paper production.

Contents

Foreword

Howard Price, Senior System Architect, System Management Group, Symbian

I have had the pleasure of being involved in the development of the OPL language since its earliest days, when Psion first provided OPL for the Organizer II, a device that had two lines of LCD text available. OPL programs were then mainly used to query and write to the built-in database system, with some mathematical, date, and string manipulation functions. Even then OPL was very popular with third-party developers, such as Marks & Spencer, which used OPL Organizer II programs at its checkout counters. In my opinion OPL was a key factor in Psion's success in the PDA market.

I wrote the Series 3/Series 3a OPL, adding support for modules, powerful graphics capability, GUI menus and dialogs, and direct access to the EPOC OS functions. By then over 90% of third-party applications were written in OPL, despite strong efforts by Psion to encourage developers to use their object-oriented C. OPL was the language of choice both for commercial applications and for shareware, including a lot of games software. A large and very active OPL developer community grew, with authors working from home or on the train to develop many shareware programs.

For the Series 5, I led the five-person OPL development team where we also added, among other things, the OPX framework. An OPX is a C++ DLL containing OPL extension functions that an OPL application can call as easily as calling OPL procedures. OPXs can also call-back to the OPL application. With OPXs, an OPL application can be as powerful and perform as well as a C++ application.

In 2002 Symbian decided to release OPL to the Open Source community, where the opl-dev project on sourceforge started in April 2003.

Why is OPL so popular? Well, here are a few reasons.

- From the beginning OPL, in all its device-specific incarnations, has been carefully designed to enable an OPL application to be fully integrated into the application architecture, to the extent that a user would be very hard-pressed to tell whether an application has been written in OPL or C/C++.
- OPL is a simple, intuitive but powerful language and can be learnt very quickly.
- Over the years the OPL development community has developed a large set of incredible OPL applications, showing just what can be done and encouraging others to try.
- The OPL development community provides OPX libraries for use by other developers.
- The OPL SDK and many other useful resources are freely available from the opl-dev project on sourceforge.
- Applications written in OPL are multi-platform – they will run as expected with very little change on any Symbian OS device. Device-specific features are generally provided in OPXs.
- Applications can be developed on the PC, or on a communicator/PDA that has a keyboard.

With the increasing efforts of the Open Source OPL community, and Ewan in particular, OPL is getting more attention than ever and I am sure OPL has a bright future, and with it I would predict that OPL will once again account for a majority of Symbian OS applications.

About This Book

In this introduction, you will learn:

- what OPL is
- the history of OPL
- what you can do with OPL
- how the rest of the book is structured.

What is OPL?

The shortest answer is that OPL stands for 'Open Programming Language' and it is a way of programming your Symbian OS Smartphone to make it do what you want!

If you've downloaded an application into your phone (for example, from the Internet) then you've already started to realize that your phone can do more than what it did, out of the box. There are thousands of applications out there that you can put on your phone. OPL will help you program your own applications that do exactly what you want them to. These applications could be for yourself and your own enjoyment or needs; they could be to help you and your colleagues at work solve a specific business problem; or you could look to putting them on the Internet and selling your software to other users.

Any programming language supported by Symbian OS can offer this to you, so why choose OPL? The first thing is that OPL itself is free. It doesn't cost you to download and use the tools needed. It is also Open Source. This means that a competent Symbian OS C++ programmer can look at the code that makes OPL work and see if they can improve it, add to it, and help maintain it... all to the benefit of OPL and the programmers who use it.

But the main thing about OPL is that it is very easy to learn, and it takes very little time to program a new application.

The History of OPL

OPL first made its appearance on the Psion Organizer II in 1984. Before OPL, all programs for Psion's machines had to be written in a very tricky, complex form of code called 'Assembler' using a PC development kit, requiring the developer to have a good, in-depth knowledge of programming.

By this time, the BASIC programming language was available for most home computers, making computer programming accessible to anyone who owned a computer. OPL was based on BASIC, but tailored for the Psion Organizer II. Users were able to write simple programs even if they didn't have the in-depth knowledge that Assembler programming required.

OPL was originally designed as a database language to access or create databases shared with the Psion Organizer II's built-in Data application, but it has evolved with each new hardware device, always aiming to maintain good backward compatibility with previous versions. This helped developers to port existing OPL applications to a new device with the minimum of effort, while at the same time giving OPL applications the ability to have the same look and feel as the built-in applications. A key requirement for OPL was to make it possible to develop applications fully on the device itself.

The power of OPL has arisen from its extensibility. OPL has supported language extensions from the beginning, via 6301 Assembler procedures on the Psion Organizer II, and now via C++ OPX procedures on phones running Symbian OS.

On the Psion Organizer II, the OPL Runtime was written in 6301 Assembler. The main functionality included loops, conditionals, one-dimensional menus, database keywords, error handling, arithmetic operators, mathematical functions, language extensions written in Assembler, and procedure files in a flat filing system. At this time, most of the applications were written for the corporate environment.

In the late 1980s, Psion launched the MC series of (laptop sized) devices. OPL was ported over to the 8086 CPU and had broadly the same functionality as the Organizer – without menus, but with dynamically loadable modules, keywords to call OS services, and input/output keywords (both synchronous and asynchronous forms).

The Psion HC was again built around the 8086 chip, but made greater use of graphical elements. In addition to the keywords added for the MC series, there were graphics keywords, the ability to call procedures by indirection, the concept of OPL applications that looked like built-in applications, event handling (for handling messages from the operating system such as switch files, close, etc.), and command line support.

The Psion Series 3 (with the advent of the 'SIBO' operating system) was released in 1991, and along with it came the first OPL Software Development Kit (SDK), giving many utilities and macros for nearly full access to the SIBO operating system services. Series 3 OPL added menus, dialogs, and the expression evaluator (used by the Calculator application).

When the Psion Series 3a came out a few years later, OPL was again upgraded and remained almost unchanged for the rest of the SIBO range (Psion Series 3a, 3c, 3mx, Siena, and the Workabout range). It added allocator keywords, a cache with least recently used procedures flushed when necessary (for up to seven times speed improvement), and digital sound support.

In 1997, OPL was ported to C++ for Symbian OS, adding pen event handling, cascaded menus, popup menus, language extensions (using OPXs), constants, and header files. Other enhancements included toolbar support and extremely powerful access to the new Symbian OS DBMS database implementation. The first Symbian OS OPL SDK was released shortly afterwards, allowing developers to develop OPL applications on a PC with the addition of a number of tools.

Symbian OS v5 in 1999 added improved color support and file recognition thanks to MIME support, amongst many other minor improvements.

When Symbian OS v6 debuted, powering the Nokia 9210 Communicator, the OPL Runtime was no longer included in the ROM of the machine, and it appeared that OPL would not be part of the Smartphone revolution. Luckily, OPL appeared as a downloadable component on the Symbian website, so OPL authors could move onto the new platforms.

OPL is now available over three major Symbian OS platforms, the Communicator range (sometimes called Series 80), Series 60, and UIQ. It has become an Open Source project, which means anyone can download the code that is used to create the runtime, the tools, and the developer environment. It is also free to use, there are no licensing costs involved to use OPL – it is truly a totally free development option.

Who is This Book For?

If you've programmed, at any level and in any language, then you'll find this book is an excellent primer for the OPL language, and you should be able to understand OPL in under a week. You should be able to start at Chapter 3, which details the tools and utilities available for OPL.

This book is primarily aimed at non-professional programmers, the IT Manager in a company that needs an application for their staff, the 'power user' who wants to do more with his phone, and anyone interested in starting programming Symbian OS phones, but wary of spending months learning the ins and outs of Symbian OS C++.

How the Book is Structured

Part 1

- **Chapter 1: Programming Principles**
  Here we look at how a computer is made up, the parts of a computer and what they do, how programming languages work, and some of the core structures of the OPL language.
- **Chapter 2: Introducing the Tools of OPL**
  Here we install the relevant SDKs, and point out the tools that are provided, and those you need to download to help you get started in OPL.
- **Chapter 3: Event Core**
  Event Core will be your first full program for OPL – in this chapter we look at the design and coding process in great detail, explaining at every step of the way what we are doing and why it is important. If you're new to programming, this will probably be the hardest chapter to comprehend, as it steps through every stage of OPL development. Once you understand this chapter, programming in OPL should be an easy experience.
- **Chapter 4: A Conversion Program: Event Core in Practice**
  Event Core is a building block for the rest of your OPL programs, but how do you expand Event Core into a new program? Here we take the core and build a real-life program; a conversion program for measurements, weights, and lengths.
- **Chapter 5: Using Graphics in an Othello Game**
  While it is possible to create a program using just menus and dialog boxes, you will want to be able to display graphics for many applications, respond to pen taps on the screen, and present a 'nice' user interface on screen. By writing an Othello program, we cover all these areas, and take a brief look at how Computer Artificial Intelligence ('A.I.') works, and how to apply this to your own games programming.
- **Chapter 6: Database and a Notepad Program**
  The final example program in the book looks at using databases to store information for your program, so it is available whenever you run your program.
- **Chapter 7: Publishing your OPL Application**
  In this chapter, we look at how to go about putting your programs on the Internet, and offer some advice if you decide to try and sell your programs online, including what you should do and where you can go to achieve this.

- **Chapter 8: Creating Applications and Installers**
  While developing these first programs, the files have been moved by hand onto the phone. This is not something you can ask an end-user to do when releasing your programs. This chapter looks at making an OPL program into a full Symbian OS application, and using Symbian OS SIS files to allow for easy installation.
- **Chapter 9: Where Now For OPL?**
  Finally we see what OPL can do in the real world, by looking at three OPL authors and what they've achieved. Al Richey, Steve Litchfield, and Malcolm Bryant all have well-respected OPL applications that they have released on the Internet.

Part 2

Part 2 contains all the reference material for OPL that you will need as you program in OPL.

- **Command Listing**
  An alphabetical list of all the standard OPL commands, their syntax, and how to use them. Includes code examples where appropriate.
- **Const.oph Listing**
  The library of constants (names that replace long numbers or strings to help make your code easier to read – these are explained in detail later) is listed in its entirety.

Author Biography

Ewan Spence studied Computing and Artificial Intelligence at Edinburgh University before discovering his first Psion PDA. Since then he has actively followed the development of mobile computing technology, and become one of the leading authorities on the OPL language of Symbian OS.

He has produced software in OPL since 1994, including the ever-popular and addictive 'Vexed' game for Symbian OS mobile phones. Since providing support for and fostering a vibrant Open Source and Freeware community for programmers through the FreEPOC Software House, Ewan has continued to help the wider Symbian community through the All About Symbian family of websites. He strongly believes that programming computers should be something that is easy, accessible, and simple to understand for every user. It shouldn't require a university degree and months of studying.

He currently lives in Edinburgh with his wife, Vikki, his two daughters, Eilidh and Mairi, and Crow, the puppet.

Author Acknowledgments

A huge amount of thanks have to go to Rick Andrews and Phil Spencer for keeping OPL alive. More thanks go to Ian Weston, Phil Northam, Edward Kay, David Mery, Colin Turfus, Martin de Jode, Lars Persson, and Peter Wikström for believing in OPL, and suggesting that an 'Introduction' book would be ''a rather good idea''.

Thanks should also be directed to Rafe Blandford, Jim Hughes, Russell Beattie, Matt Croydon, Monty, Mobibot, Robin Talboom, Jordan Holt, Andy Langdon, Hayden Smith, Craig Setera, Matthew Langham, Frank Koehntopp, and everyone else involved in All About Symbian and Mobitopia who've had to put up with my promotion of OPL (and my spelling) for several years.

Steve Litchfield, Al Richey, Jon Read, Martin Harnevie, Andy Harsent, Martin Dehler, Domi Hugo, as well as Malcolm Bryant, Adrian Pemsel, Martin Guthrie, and all the other guys at FreEPOC must be mentioned for not only being better programmers than me, but for letting me look at their source code and learn from it.

And a series of special mentions go to... Rael Dornfest, for providing a shot in the arm that showed me OPL was actually going somewhere. Jerry Sadowitz, for being the second greatest card magician alive. Suw Charman and Jeannie Cool, just for being around. Kenton Douglas, for an inordinate amount of time off work. Danny O'Brien and Dave Green, because they get thanked in everything and I don't want to break the chain. Janne Jalkanen, for the Go lesson and the subsequent alpha application in OPL. Hector and Russell, the Kiltmakers. And finally Ed, Frankie, and the Hawkins brothers, for the music that this book was (mostly) written to.

I know I've probably missed a bundle of people involved in OPL, but thanks go to them as well. Get in touch with me and if there's a second edition, I'll add you in!

But the biggest thanks of all go to Vikki, Eilidh, and Mairi. For being with me in my life, spending time with me, and putting up with everything I had to do to write this book – and everything else. . . I can never thank them enough.

Symbian Press Acknowledgments

Symbian Press would like to thank Ewan for his steadfast dedication to the cause of OPL. Thanks too must go to young Phil Spencer for being a veritable cornucopia of information in times of need. Eternal gratitude to Phil n' Freddie for their master class in publishing, no one can underestimate the value of a few choice words, even when spoken through the froth of a pint.

Cover concept by Jonathan Tastard.

About the Cover

The mobile phone has traditionally connected the mouth to the ear – at Symbian's Exposium 2003, Symbian introduced the concept of Symbian OS, enabling a new generation of connected communications devices by connecting the mouth to the ear to the eye. To realize this vision, the mobile phone industry is working together through Symbian to develop the latest technologies, support open industry standards, and ensure interoperability between advanced mobile phones as networks evolve from 2.5G to 3G and beyond. . .

Symbian licenses, develops, and supports Symbian OS, the platform for next-generation data-enabled mobile phones. Symbian is headquartered in London, with offices worldwide. For more information see the Symbian website, ***http://www.symbian.com/***. 'Symbian', 'Symbian OS', and other associated Symbian marks are all trademarks of Symbian Software Ltd. Symbian acknowledges the trademark rights of all third parties referred to in this material.

Part 1

1

Programming Principles

In this chapter you will learn:

- the essential parts of a mobile computer and how they relate to each other
- why there are many languages to program a computer with, and why they are all different
- the main elements of computer coding; so-called 'variables', the `DO...UNTIL` loop, the `WHILE...ENDWH` loop, and the `IF...ENDIF` structure.

1.1 The Language of Computers

To program your Symbian OS Smartphone the first thing to remember is that it is a computer. Sure, it may be a lot smaller than the one on your desk, and it has a phone built in to it – and it may or may not have a full QWERTY keyboard – but nonetheless it is a computer. And to program a computer, you need to speak its language.

1.1.1 Storing Information

A computer is a digital machine, which has a language that consists of two characters, 1 and 0. That's it. These represent the two electrical states that each tiny portion of the computer can have (on or off). This is called a bit, and it is the smallest structure in the language of a computer. A memory chip can hold millions and millions of these states, and like any language, collections of these pieces (individual 'letters' if you will) can be grouped together to make more complex 'words'.

If you have a collection of 8 bits (eight 1s, eight 0s or a mix of them) then this is called a byte. A byte can represent any number from zero (00000000) to 255 (11111111). Why does this equal 255 and not eleven million, one hundred and eleven thousand, one hundred and eleven!?

Well, that is a subtle complexity of the way this special 'binary' language is interpreted by computers. If you don't want to know, please feel free to skip this bit – it's in no way essential to your understanding of OPL. If you're just a little curious then there are various excellent sources on the Internet which explain the binary system in detail – for example, search for 'binary numeral system' on ***www.wikipedia.org/*** to learn more.

When counting bytes, it starts to get unwieldy when you get to around a thousand bytes; which is where the kilobyte comes in. A kilobyte is 1024 bytes, and is commonly written as 1 K (or 1 Kb). And when we get to a thousand kilobytes, we have *another* term: 1024 kilobytes equals one megabyte, written as 1 Mb. You'll probably be aware that your Smartphone has a certain number of megabytes of memory. This is where that number comes from. So if you have 4 Mb of memory, you can store thirty three and a half *million* ones and zeros. These bits, bytes and kilobytes represent information – not readable to you as a user, but basically *all* the phone itself understands.

1.1.2 Processing Information

What can we do with this information? Well, we can't do much with it. But the phone can – it can take this information from its *memory* and then process it somehow. This happens in the Central Processing Unit (CPU), more commonly just called the 'processor'.

The CPU reads information from the memory and follows the instructions that it finds there. We said above that a group of bits makes up a 'word'. The length of this word (8 bits, 16 bits, 32 bits or more) is an indicator of how complex a CPU is. Home computers of the 1980s were based around CPUs that could read 8-bit words. Nowadays, desktop computers are generally 32-bit or 64-bit. Your Symbian OS Smartphone is 32-bit, so each word is made up of 32 bits.

Inside the CPU there is a 'dictionary' of all the unique words that the CPU understands, and what it should do when it reads in one of these words from memory. This is the essence of a computer program; a list of things to do.

1.1.3 Talking to the Users

It's all well and good having the processor reading information from memory, but how do we know what's happening? Or tell it what information to read? This is where inputs and outputs (I/O) come into play. I/Os are how computers are told what to do (input) and how they report back on what they have done (output).

The most common form of input is a keyboard; but inputs can also include things like touchscreen display, a microphone, a light sensor, in fact anything that passes information into the computer. Information from the Internet is regarded as an input as it is passed *into* the computer.

Talking to the Internet is also an output, as your computer needs to give information to the Internet (e.g., downloading a web page). What is displayed on the screen is an output, as is a speaker, a buzzer, a flash on a digital camera. . .anything that comes out of the computer.

Memory can be regarded as something that does I/O operations to the processor: it passes stored information to the processor (input) and takes the responses or results from the processor and stores that information (output).

1.1.4 Keeping a Note of Things

Memory on a computer is not like the memory you and I have. Just because something is put in memory does not mean it stays there for ever. A computer's memory is like a temporary workspace. The processor will copy a program into memory from a storage device. Some commands may ask the processor to read information from storage for the program to use (for example, a list of phone numbers). Any changes to information will need to be made to the equivalent information in storage if it is to be kept permanently.

And yes, a storage device is an I/O device, but it's an important one, so gets recognized in its own right.

1.1.5 Putting it All Together

In abstract form, your computer looks like the model in Figure 1.1.

The Life of a Program

When a user runs a program, they start it off with an input (maybe typing a command or selecting an icon). The Central Processing Unit, (commonly known as the processor or CPU) recognizes this command from its dictionary and reads the program itself in from the storage device, copying it into memory where it can be accessed directly by the processor. The processor can only read from memory directly, hence this intermediate step is almost always required. The processor reads these commands from the memory one at a time, looking each one up in its dictionary. Some of these commands may be to output information (such as the result of a calculation). When there are no more words to read in from the memory, the program is finished and the processor waits for more input.

1.2 Speaking the Language

Now we know *how* a program works, how do we *write* our first program? Well, when computers first came about, everybody programmed directly in binary. That is, they actually wrote down and manually inputted all the

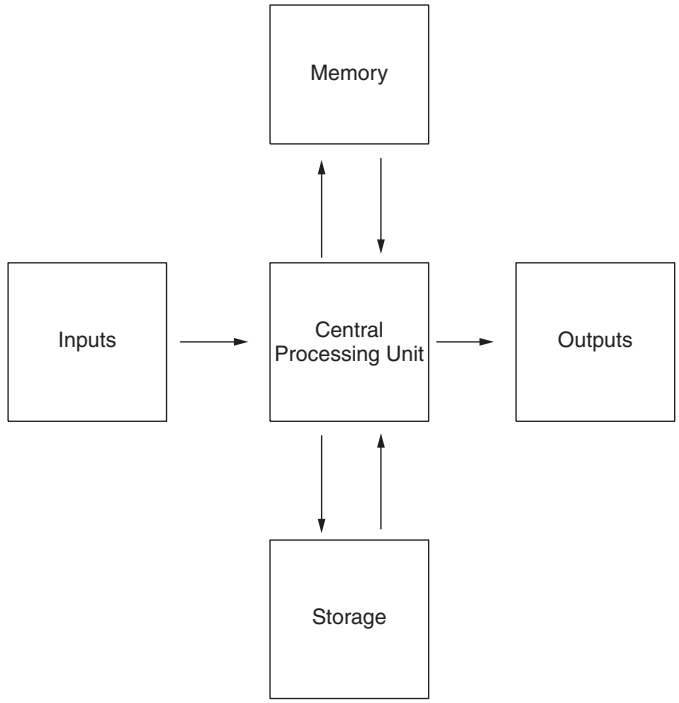

Figure 1.1 Abstract computer model

ones and zeros themselves, by looking up what they wanted to do in a human dictionary of commands (similar to the one inside the processor). This is called, naturally, machine code, because it is the code of the machine itself. If you work at this level, then you are said to be working at a low level – the lowest of all, in fact – hence the term *Very Low Level Programming*.

1.2.1 Assembly Language

After a short period of time, one programmer came up with a smart idea. Instead of writing down each word of 1s and 0s, he wrote down the list of commands so that one command matched one computer word. So 10101010 could also be written as `load hl`. These were called mnemonics, and while still arcane and hard to understand, they did mean that it was much easier to read and write computer code.

But the processor could still only recognize 1s and 0s. So a program was written that read in this list of mnemonics, looked up the database, and outputted 1s and 0s that could be read by the processor. This program could take something that was easy for users to read, and assemble something that was easy for computers to read. Hence, these mnemonic words became known as Assembly Language. No longer did

everyone have to work at a very low level; they could now work at 'just' a low level.

This is the principle of writing source code, and then translating it into something that is readable by the computer (called object code) before storing or running it. It still required a lot of knowledge about how the processor worked, and while still very hard to program in, it made life a lot easier.

1.2.2 Climbing up to Higher Languages

Nowadays, there are a *lot* of languages out there, some of them work at a low level, like Assembly, but most people prefer to work in languages that are easier to read and offer other advantages.

Symbian OS offers developers many choices of development language including native C++, Java, Mobile Visual Basic ('Crossfire' from Symbian Partner AppForge) and, of course, OPL. The two most widely used languages are currently C++ and Java (specifically the Personal Java or MIDP implementation).

A simple diagram (Figure 1.2) illustrates the move from lower-level languages to higher-level ones.

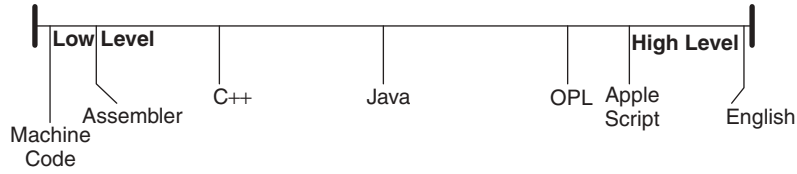

Figure 1.2 Transition from lower-level to higher-level languages

C++ – The Halfway House

You hear that C++ is a very powerful language. This means that it can do a lot of I/O operations, and as a programmer you have to be able to specify exactly what has to happen. This means you have a large amount of control over what you can make the processor do, but you also have to understand the effects of everything.

A good C++ programmer needs to know almost everything about every subject that we might encounter – controlling the screen, communications devices, memory access, etc. C++ itself also offers advantages to programmers in terms of 'code re-use' to save them re-implementing lots of code in multiple programs. This is a classic trade-off in programming – very often, the more power a language offers you, the steeper the learning curve.

Java J2ME (MIDP)

One of Java's strengths is that once you have written one program, it should be able to run on any computer that contains Java. Why is this a

strength? Because most different types of computer have a different list of instructions in the processor. This is why programs written in C++ and other low-level languages that run on one computer (e.g. an Apple Mac) will not run on another computer (e.g. Windows-based PC). The 'price' you pay for this is more limited access to some system functionality compared to C++.

BASIC

One of the earliest 'high-level' languages was BASIC. Commands in BASIC were very close to readable English, and meant that the learning curve associated with the low-level languages was not present. Many of the personal computers available at the start of the home computer boom in the early 1980s shipped with BASIC installed on them, and this led to a huge cottage industry of curious programmers programming their machines to do whatever they needed them to do.

1.2.3 Compiled Languages

We've already seen that the principle of source code being compiled to machine code is present in Assembly Language and C++, but higher-level languages (such as Java) work slightly differently. The source code for these languages is compiled into an intermediate form, and this code is the object code.

The object c is read in by another program. This program can take the instructions in the object code, and interpret these into the correct commands that need to be sent into the processor. This program can be called an interpreter, or a runtime.

Runtimes are usually device-specific, but are written in such a way that the object code can be read by *any* runtime, no matter what computer the runtime is running on. This is the principle of write once, run anywhere. Many high-level languages have this capability to some extent.

1.2.4 The Trade-Off

So why doesn't everyone just use the highest-level language possible? Two considerations: speed and access to functionality.

Compiled high-level languages are much slower than lower-level languages such as C++. Each command in the object code has to be looked up and translated into an instruction by the runtime as you go through your program. In lower-level languages, the code is already in the form the processor can understand, so there is no overhead to 'translate' or 'interpret' it.

In order to ensure the object code is universal, it must conform to some kind of standard – thus you might not have access to all of the

functionality that is available to lower-level programming (for example, the Bluetooth I/O device) if the current standard does not specify this. If your runtime *does* allow you to access these features, it may be much slower than accessing it from a low-level language.

1.2.5 Where does OPL Fit in?

OPL, in a similar way to Java, is an interpreted language that needs a special 'OPL Runtime'. This runtime can be packaged with every OPL application or downloaded separately from the Internet (we'll show you where when we gather all our tools in the next chapter). If the runtime is included in the actual package, the file size will be much larger, so it is common to provide a download reference in your documentation on where to download the runtime.

The OPL Runtime is written in C++, to make sure the speed penalty of using a high-level language is minimized. OPL can be extended to access the device functionality through a feature called OPXs. An OPX is an 'OPL eXtension', and is a small piece of C++ code that can be loaded into your phone. OPL can then call this extension with a simple line of code.

1.3 Learning the Vocabulary

Now we've had a look deep inside your phone, at how it works and the basics of what a program is, we can start looking at OPL and how it works.

Like any language, OPL has a grammar that you need to follow to be understood. You have words (commands) that have to be followed by certain things to make them work. These make up lines of code (sentences) and these lines of code can be grouped to make procedures (paragraphs).

From now on, we're only going to concern ourselves with how the computer reacts to the OPL code you write. Remember that once it is compiled, the runtime will do the work required to allow it to talk correctly to the processor, and OPL developers never need to concern themselves with this.

1.3.1 Procedures

When an OPL program is run, the first procedure (here called `PROC Main:`, though commonly the first procedure is named after the program) is opened. The lines are then read and processed in the order they are listed, until the `ENDP` (end of procedure) command is reached, at which point the program stops itself.

Within a procedure, you can call another procedure. Do this simply by typing the name of the procedure you want to run next. All procedure names must have a colon after them (:) in both the `PROC` command and when calling that `PROC` from inside the code.

```
PROC Main:
        SetupApp:
        DoSomethingNice:
        SaveStatus:
ENDP
PROC SetupApp:
        rem Do something interesting here
ENDP
etc...
```

In this example, `PROC Main:` is opened, it calls three other procedures in order, then reaches `ENDP` and the program closes. Note that each procedure must start with `PROC <a name>:` and end with `ENDP` on separate lines. When the end of the first procedure is reached, your OPL program stops. The only way to run other procedures is to call them in this way.

Procedure names cannot have spaces in them, so you'll see that each new word is signified by a capital letter. While OPL is not case-sensitive, this is the recommended style for writing code. If you follow this style, it makes it easier for you (and other programmers) to read your code.

1.3.2 The Remark Statement

Speaking of making code easier, you'll see in the example above that we have a line with a new command.

```
rem Do something interesting here
```

`rem` stands for remark, and it's a powerful statement for anyone reading the code. You see, the `rem` statement does absolutely nothing. The interpreter ignores anything after the `rem` statement on the same line, so you can use it to add notes, thoughts, and descriptions throughout your code. For example, a procedure may have something like this at the start:

```
PROC WhereIsTheCursor:
        rem This routine calculates the cursor position
        rem FooX%% represents the temporary x co-ordinate
        rem FooY%% represents the temporary y co-ordinate
etc...
```

Not only are these `rem` statements useful when you come back to the code in six months' time and can't remember what something is for, they are also useful if other people are going to read your code.

1.3.3 Variables

A variable is something you want your program to remember for a certain amount of time – either throughout the *entire* time the program