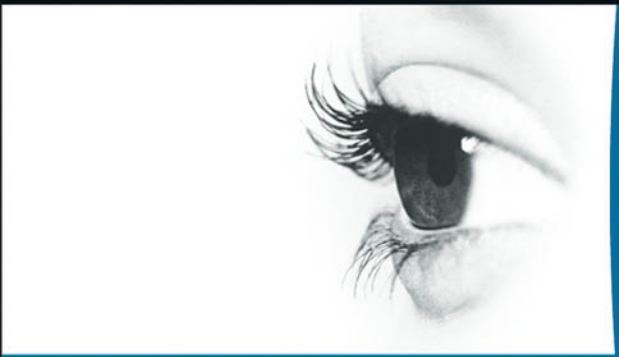
WILEY

Rapid Mobile Enterprise Development for Symbian OS



An Introduction to OPL Application Design and Programming

symbian

Ewan Spence

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

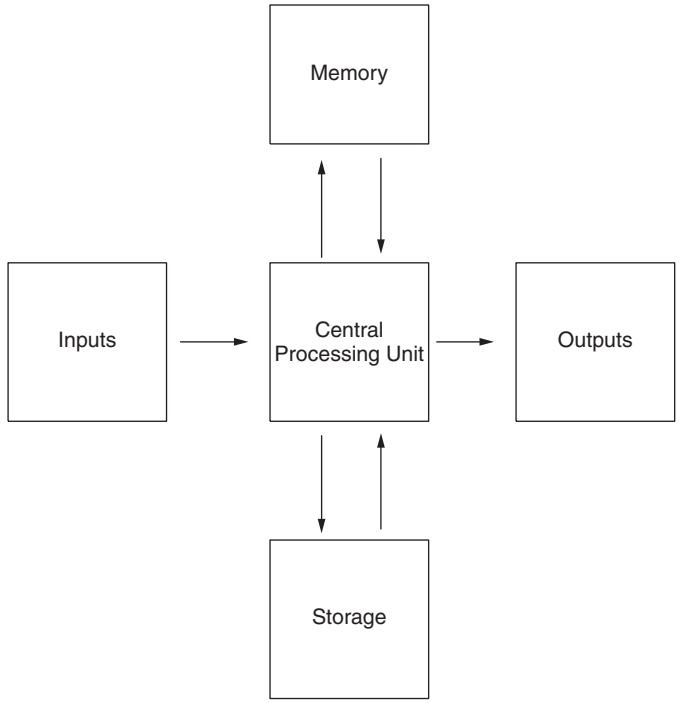


Figure 1.1 Abstract computer model

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.

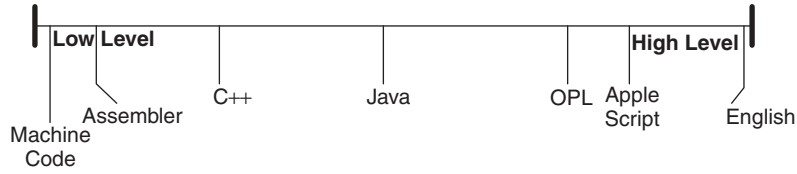


Figure 1.2 Transition from lower-level to higher-level languages

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

is running, or just during one particular procedure. A variable can be a small number, a big number, or a string of letters and numbers. Each of these variables is given a name, and a small sign to indicate what type of information it represents. You should also remember these names cannot have a space in them either, so don't forget to use capital letters for YourVariableNames.

Types

- A small number (called a 'short integer') is followed by a % sign. For example, `HighScore%` is a short integer that stores a number, and can be referred to as `HighScore%` in your code. An integer has to be a whole number (i.e. you can't have `HighScore%=2.5`, but you can have `HighScore%=2` or `HighScore%=3`). The maximum value a short integer variable can store is 32767, the minimum −32768.
- A larger number (or 'long integer') is followed by a & sign. For example, `BigNumber&` is a large integer that stores a number, and can be referred to as `BigNumber&` in your code. Just like small integers, these large integers can only be whole numbers. The maximum value a long integer variable can store is 2147483647, the minimum −2147483648.
- Numbers that contain decimals or fractions are implemented as 'floats' (short for 'floating point numbers'). These are easiest to code – you simply type the variable name with no suffix and a float type is implied, for example `MyFloat`. The maximum value a float variable can store is 1.7976931348623157E+308 and the minimum is 2.2250738585072015E−308.
- A string of letters and numbers is called (funnily enough) a string, and is followed by a $ symbol, For example, `Name$` could be set to contain "Ewan Spence". A string variable can be a maximum of 255 characters. Note that a string can hold any valid characters, so could also represent a number – for example, `Name$="1138"` is a valid string. You can't, however, do any arithmetic on a string like this.

You may have a simple question at this point regarding the different types of numerical variable – ''if float lets me store any type of number and to a much larger size than the other two, why don't I just declare all my numbers as floats?'' The answer is simple – floats use more memory. Think of variables as 'boxes' or 'pigeon holes' – the bigger the box, the more space (or memory) it requires. It is therefore wasteful to always use a float.

Defining Variables

Before you can use a variable, you need to have told your program that you are going to use it by defining it. Think of it as reserving a little

space in memory to store the information. There are two ways of defining a variable.

A GLOBAL variable is a variable that any procedure in the program can use, at any time.

A LOCAL variable is a variable that only the specific procedure in which it is defined can use. When you leave the procedure, the little space in memory for the number is destroyed. Local variables are good for temporary counters and bits of info that are only needed for a few moments (rather like a scrap of paper).

The command to define (or 'declare') a variable needs to be the first command of a procedure (although remember that rem comments do not count as commands, so these are acceptable before your variables). GLOBAL definitions must be the first lines in your first procedure, and LOCAL variables must be the first lines in the relevant procedure.

You can put more than one definition on a line – simply separate them with a comma, and they can be different types of variables.

```
PROC Main:
GLOBAL HighScore%,BigNumber&,MyFloat
```

And. . .

```
PROC LocalExample:
LOCAL HouseNumber%,Name$(20)
```

Note that after defining Name$ we have a bracketed number. This tells the computer the maximum length (in characters) this particular string will ever need to be. You need to define the maximum length of all strings or when you translate the program to run, you'll get an error message.

All numerical variables do not need this length definition, as they already have a 'largest number' limit implied (see above).

Setting Variables

Variables are easy to set. Simply use the equals sign. For example:

```
HighScore%=56
BigNumber&=1383512467
MyFloat=559.8798
Name$="Ewan Spence"
```

Setting a variable should be done on a separate line for each command. Note that when setting a string, you have to use quotation marks. These don't appear as part of the string, they are used in the source code to show where the string starts and ends.

Once a variable has been set, you can of course change its value.

For example:

```
HighScore%=56
HighScore%=58
```

results in `HighScore%` first being assigned the value of 56, then the value of 56 is thrown away and `HighScore%` is assigned the value of 58 instead. You can also have a variable set to equal the value of another variable:

```
NewHighScore%=LastScoreInGame%
```

1.3.4 Arrays

One other tool you have in variables is an array. An array is a list of things. For example, if we wanted to have a table of 10 numbers, we could do:

```
GLOBAL Table1%,Table2%,Table3%
```

This would work, but imagine now doing a list of a hundred things – that would be a lot of typing! This is where an array comes in useful. If you think of variables as being 'boxes' in memory, each with a name (the variable name), an array is a line of boxes, and that *line* has the name, each box has a number. It's a bit like a spreadsheet – the array variable is 'Row A' and the individual 'elements' are columns 1, 2, 3, etc. on that row.

To illustrate, let's make an array called `Table%` to handle those hundred things. Firstly, as with any variable, we need to define it:

```
GLOBAL Table%(100)
```

This will create an array of 100 items (the number in the brackets) and call it `Table%`. The array will hold short integers, signified by the % sign.

The first 'box' can be addressed as `Table%(1)`, the next as `Table%(2)`, and so on. Smart-eyed readers will see that this bears a striking resemblance to how we define a string – which has a maximum length. This is because each character in a string effectively takes up one 'box' of memory.

So can we make an array with strings? Yes, we can. We will need to define the maximum length of the string inside the brackets, and we also need to say how large the array will be (how many strings are in the array). This becomes a second number inside the brackets, separated by a comma.

So defining a string array looks like this:

```
GLOBAL NameTable$(100,20)
```

which is an array of 100 strings, each a maximum of 20 characters long, and the array is called `NameTable$`.

So why use `Table%(100)` and not `Table1%`, `Table2%`, `Table3%`, and so on? Because if you have another variable (e.g. `LookUpThisNumber%`) then you can simply refer to `Table%(LookUpThisNumber%)` to get its value, rather than having to actually write in code ''Well, if `LookUpThisNumber%=1`, return `Table1%`. If `LookUpThisNumber%=2`, return `Table2%`, and so on'' – which, for 100 elements, would be very, very wasteful!

We will illustrate this further shortly.

1.3.5 Constants

Constants are exactly what they sound like – things that don't change! They are a bit like variables (so they can be numbers or strings), but they can never change value. You can use them so code is easier to read. Rather than having to type the number 103782376 every time you need to use it (assuming it is something that appears a lot in your program), you can put the following at the start of your code:

```
KBigNumber&=103782376
```

Each time you need to use this number, you can write `KBigNumber&` instead. By convention, you should prefix any constant name with `K` to help differentiate it from normal variables. Constants should be defined at the start of your source code, before the first procedure.

```
CONST KBigNumber&=103782376

PROC Main:
GLOBAL HighScore%,HighScoreName$(100)
etc...
```

Once you see some example code, constants should be a lot clearer. One thing to point out now is that there are a lot of default constants for things you will use a lot. If you have the line

```
INCLUDE "Const.oph"
```

at the top of your code, then these default constants can all be used in your own code too. A list of these constants can be found in the OPL Documentation.

1.3.6 Loops

Do...Until

Loops are a great way to make your program do something over and over again, maybe with a few changes. Look at this code:

```
PROC DoUntilLoop:
LOCAL Foo%
        Foo%=0
        DO
                Foo%=Foo%+1
                Table%(Foo%)=Foo%
        UNTIL Foo%=100
ENDP
```

To make sure you understand this code, let's look at it line by line. The `DO...UNTIL` loop is a primary building block of source code, because lots of programming involves repetition. Firstly we create a variable to act as a counter, in this case `Foo%`. This is a `LOCAL` variable so will only be accessible to the `DoUntilLoop:` procedure, and the space reserved for it in memory is reclaimed once the procedure is finished.

You'll see me use `Foo%` (and `Gnu%` and `Zsu%`) as a temporary variable name a lot. If you do see it, you can assume it's a counter or another temporary local variable.

First of all we set the temporary variable `Foo%` to be zero. We then reach the start of the loop (`DO`). The code will then `DO` whatever comes up next – here, add one to the current value of `Foo%` first of all (`Foo%=Foo%+1`) and then set the array element of the `Table%` array to be equal to the new value of `Foo%`. We will keep doing this `UNTIL` the value of `Foo%` is 100. Thinking ahead, what this code means is that at the end of the loop, our `Table%` array will have 100 sequential numbers in each box, and the box position will match the value.

While...EndWhile

`DO...UNTIL` is the primary loop that you will see in this book. One thing you need to note is that the code inside a `DO...UNTIL` loop will *always* be carried out a least once – think for a minute to work out why this is the case (*Hint*: remember OPL code is executed line-by-line in order).

There will be circumstances where you might want to check the condition *before* the code in the loop is reached. If this is the case, you would use the `WHILE...ENDWH` (EndWhile) construction.

```
PROC WhileEndWhileLoop:
LOCAL Foo%,Table%(100)
```

```
    Foo%=0
    WHILE Foo%<100
        Foo%=Foo%+1
        Table%(Foo%)=Foo%
    ENDWH
ENDP
```

Here, the code inside the loop is only carried out if the comparison in the WHILE statement is true (i.e. the reverse of the UNTIL statement). The code in the loop will continue to be run until the statement is false. At this point the commands after the ENDWH will be read. Here, we are checking for Foo% being not equal to 100 (thus the WHILE condition will become false once Foo%=100).

You'll note that although these two loop styles work logically in different ways, the end result is the same. This is something that happens in code a lot, where you can achieve the same end result through different methods. Don't worry if you do something slightly different to somebody else – the end result is the most important thing to consider.

Break

So we can come out of these loops at the start of the loop, or at the end of the loop. What happens if we want to come out of the loop in the middle of the loop code? We can use the BREAK command.

When your program reaches a BREAK command, it will jump to the first line of code after the end of the loop (either the UNTIL or ENDWH command). BREAK statements are usually inside an IF statement, as you shouldn't be using the BREAK statement very often – it's mentioned here for completeness only.

1.3.7 Decisions, Decisions

If...Else...Endif

The IF...ELSE...ENDIF construct is the third great building block of computing languages.

At many points in your program, you'll have to check something and then execute different commands depending on the outcome. To check things you use the IF command.

```
IF Guess%=1
        GuessRight:
ELSEIF Guess%=2
        GuessClose:
ELSE
        GuessWrong:
ENDIF
```

Hopefully, you should by now be able to make a good guess (no pun intended!) as to what this code is attempting to do.

If the variable `Guess%` equals 1, then go to a procedure called `GuessRight:`. If `Guess%` equals 2, then go to a procedure called `GuessClose:` If `Guess%` is anything else, then go to a procedure called `GuessWrong:`.

Note that one of these options has to be chosen. Using `ELSE` on its own means that if all the other tests have failed, then do this. It's a safety net in some cases, as the program must go somewhere. The last resort here is to call `GuessWrong:`.

1.4 Summary

This chapter started by assuming you knew nothing about the inner workings of a computer. We've taken you through the basic principles, both of how a computer works and how different languages came into being. We looked at their strengths and weaknesses, and why certain languages are a better choice than others.

Finally, we looked at the main building blocks of the OPL language:

- Using constants to make code easier to read
- Storing information using variables
  - small numbers (short integers)
  - large numbers (long integers)
  - numbers with decimal points (floats)
  - textual information (strings)
- Using arrays for large collections of variables
- Repeating sections of code by using loops
  - the `DO...UNTIL` loop
  - the `WHILE...ENDWH` loop
- Making decisions
  - `IF...ELSEIF...ENDIF`

2

Introducing the Tools of OPL

In this chapter you will learn:

- the files and file extensions that you will need to be familiar with to program in OPL
- the tools that are available to help you program in OPL, where to find them, and where to get help in installing them
- the process of creating an OPL program, what tools are required at each stage, and how these tools are used
- taking these ideas and creating your very first OPL program!

2.1 Parts of OPL

Like any computer language, OPL is made up of many different parts that must all come together. We've looked at the basic syntax and structure of the OPL language in the previous chapter, and here we'll look at physical organization of your code during development. This means file names and extensions, what they are for, and how they relate to each other. Some of these may be familiar if you have already developed with other languages on Symbian OS.

2.1.1 Source Code (.tpl or .opl)

Source code is what you will regard as your program. It is a plain text file of words, numbers, and symbols that you will work on and edit in your text editor. This text is then read by a compiler (or, in the case of OPL, a 'translator') – a program that takes this easily readable code and turns it into something that a computer can understand and relate to.

Writing computer code, even in a high-level language such as OPL, requires you to be precise in your language, and all the commands and statements must be typed in exactly. The OPL translator will check that what you have typed in follows these rules.

It is also possible to write source code directly on your Symbian OS phone using the Program application (if available). While this is something that is comfortable to do on the Communicator devices entering more than a few lines of code on a Series 60 or UIQ device will take some dedication. Using a bluetooth keyboard such as the Nokia SU-8 W or the ThinkOutside Bluetooth stowaway keyboard can be a great help here.

Source code written through the Program application on the phone will not be directly readable on your PC text editor, as there are a few Symbian OS flags and unique identifying numbers embedded in the source code file because it will be saved as a 'native' Symbian OS document.

If you enter source code on your phone, it is recommended you give it the extension .opl. Whereas, the .tpl extension should be used on a PC (the 't' signifies a pure text file, as opposed to one with Symbian OS flags and UID numbers).

If you wish to move source code between your Symbian OS phone and your PC, you should use the `Import text` and `Export text` options that are present in the Program application.

2.1.2 Header Files (.tph or .oph)

A header file is used to make reading your OPL code easier. For example, as discussed in the previous chapter, if you use the number 99 constantly in your program to represent someone's house number, your header file can have a line reading something like:

```
CONST KHouseNumber%=99
```

Now, in your code when you want to use 99, you simply write `KHouse-Number%`. This makes code a lot easier to read, and also means that if you need to change the house number, you only have to change one line in the header file, as opposed to every occurrence of 99 in the source code. Again, the .oph extension applies to headers saved in Symbian OS format on your phone, with .tph reserved for the plain text versions saved on the PC.

The main const.oph is a standard header file and can be included in any OPL program as it is part of the standard package.

2.1.3 Compiled Code (.opo)

Once your source code has been passed through the OPL translator, this code is output to another file. It is now called 'object code', to distinguish it from the raw source code. Unless otherwise specified (see below), your object code file will have the same filename as the source code file, but a different extension. This extension is .opo. So `MyCode.tpl` would translate into `MyCode.opo`. The .opo file produced will be the same

regardless of where it is produced (i.e. either by translating your code on the phone or on the PC).

Once sent to your phone, this .opo file can be selected and run. You can do this through the built-in file manager on Communicators or using a third-party file manager on the UIQ and Series 60 platform – see below for some unsupported suggestions.

2.1.4 Complied Code (.app and .aif)

You can also choose to translate your source code so it appears as an application, with an icon on the main launcher screen (System screen) of your Symbian OS phone. This is done by adding the `APP...ENDA` commands at the start of your source code. When you translate your source code into an .app, an Application Information File (.aif) is also created. This is used by the smartphone to recognize an application and display it on the main System screen, as an icon which can be launched. OPL applications are covered in more detail in Chapter 8.

2.1.5 Graphics File (.mbm)

What is a computer program without pictures, icons, and graphics to look at? The Symbian standard graphics file is the .mbm, which stands for Multiple BitMap, and it is just that. It allows you to hold a 'book' of images and use them in your OPL program. They are created on your PC from standard Windows bitmaps. Graphics handling and .mbm files are covered in more detail in Chapter 6.

2.1.6 Installation File (.sis)

One of the advantages of Symbian OS is that it is incredibly easy for end-users to install applications. All the application files are packaged up in a single distribution file with the extension .sis. This file can be run on your PC, and the PC Suite will extract the application files and place them in the correct locations on the Symbian OS phone.

You can also select .sis files that have been downloaded onto the Symbian OS phone, and the installer will extract and place the files where they are required.

If you have created your OPL program to be an application, then .sis files are the expected distribution method, and you should not provide the files separately. The creation of .sis files is covered in more detail in Chapter 8.

2.2 Organizing your Projects

Your OPL program can be built up of many small files that you will need to keep track of. The system I use is to have a root folder on my PC called

OPL, and then give each program 'project' its own name. Inside that I have five folders containing the required files. I'll be using this structure throughout this book, but it is not the only way to do it – if you find a way that works for you, then go for it.

```
Root:   C:\OPL\<project name>
                        \Source\
                        \Source\Archive\
                        \Compiled\
                        \Graphics\
                        \SIS\
```

- Source folder
  Used to store the text format source code files. The Archive subfolder holds previous versions of the code so I can look back if I delete something or introduce errors.
- Compiled
  Move any compiled .opo files here for long-term storage.
- Graphics
  Holds the bitmaps and the resulting .mbm files.
- SIS
  Any file that needs to be put in the installation .sis file is moved here. This folder will also contain the script text file that will be needed to assemble the .sis.

2.3 Gathering Tools

To develop in OPL, you will need to have the relevant software tools. These are all available over the Internet (some of them are on the accompanying website ***www.symbian.com/books/rmed/rmed-info.html***). These will need to be installed on your PC before we move on to creating our first OPL program, "Hello World".

2.3.1 Text Editor Application

You will need a program to enter the source code into a plain text file. Notepad ships as part of every Windows installation and you can (if you wish) use Notepad for all your source code needs. But there are other text editors out there that are geared towards programmers entering source code on their PC. Three of the more popular choices are TextPad (***www.textpad.org/***), Crimson (***www.crimsoneditor.com/***), and Source Edit (***www.sourceedit.com/***). All OPL needs is the source code saved as an ASCII (ANSI) text file. The OPL translator on the PC does **not** use Unicode text files.

2.3.2 C++ SDK

Unless you have good reason, you should use the default paths when installing the SDK, and install all the optional components. This will make it easier to use the command line tools and follow the examples in this book.

Series 60 and Communicators

The regular C++ SDK can be used. Links and information can be found on the accompanying website ***www.symbian.com/books/rmed/rmed-info.html***. Alternatively, the SDKs can be downloaded at Forum Nokia (***http://forum.nokia.com***).

UIQ

While you are not going to be using C++ to code your programs, the UIQ C++ SDK has many of the underlying files and tools that need to be present on your PC to develop in OPL. The easiest way to have these in place is to install the SDK from ***www.symbian.com/developer/sdks_uiq.asp***. There is more than one download type for the UIQ SDK, so you should make sure that you download the Metrowerks Codewarrior UIQ SDK. Links and information can be found on the accompanying website, ***www.symbian.com/books/rmed/rmed-info.html***.

What if I Want to Program for Communicator or Series 60 Phones?

There is nothing to stop OPL programs being compiled using the UIQ SDK running on non-UIQ, Symbian OS phones. Because OPL is file-compatible across the range you can compile with any of the SDKs, and the resulting object code should run on other phones. It is recommended you install the SDK for the primary machine you will be programming on.

As we work through our examples, we'll show you how your OPL code can be written so you only have to maintain one version that will run over all the OPL runtimes for Symbian OS.

For the purposes of this book, we'll assume you are using only the UIQ SDK and you have not altered the files or folder names from the default values.

2.3.3 OPL Developer's Pack

Symbian OS licensees do not ship OPL as part of the standard SDKs, so you will need to add these elements in yourself. The OPL SourceForge Project page will provide you with the relevant OPL Developer's Pack.

Download: ***http://opl-dev.sourceforge.net/***. You'll need to copy and place the files in the correct directories manually, but this is covered in detail in the accompanying documentation.

2.3.4 The OPL Runtime

The OPL Runtime for your phone is a standard .sis file and can be installed just like any other program. The file will be found on the SourceForge site along with the Developer Pack. Assuming you have the PC Suite installed for your phone, double clicking the .sis file will start the PC Suite's installer program and place the runtime onto your phone.

2.3.5 Command Line Tools

Now the above elements are in place, you have access to three tools that are run from the command line on your PC.

OPLTran (the PC-Based Translator)

OPLTran is the primary tool for taking OPL source code and creating object code. The command syntax is:

```
OPLTRAN <sourcefile> <flags> <output file> <flags>
```

When installing the UIQ C++ SDK your PC's `PATH` variable will be updated, adding the directory that holds all the command line tools. This means you can use the tool from any directory on your PC.

Open up the command line on your PC (use Start | Run 'cmd.exe' on Windows 2000/XP, or 'command.com' if you're using Windows 98). You need to navigate to the directory holding your OPL source. If you are using the suggested folder structure, this will be `C:\OPL\Tim\Source\` (assuming the project is called 'Tim'). To translate the Tim source code file (`Tim.tpl`) you would type in:

```
OPLTRAN Tim.tpl Tim.opo
```

The resulting .opo file can then be copied to `C:\OPL\Tim\Compiled\`. Alternatively, you could use:

```
OPLTRAN Tim.tpl ..\Compiled\Tim.opo
```

This will copy the compiled file to the compiled directory if you are using the recommended folder layout.

OPLTran (for File Conversion)

As noted earlier, OPLTran requires your source code to be an ASCII (ANSI) text file (7-bit character representation), not Unicode text. Symbian OS uses Unicode throughout, so if you are moving a text file from the phone (e.g. an export of source code from the Program application) then you

will need to use the convert function of OPLTran:

```
OPLTRAN Tim.tpl -conv
```

This will take our source code file called `Tim.tpl`, and change the format. If it is ASCII (ANSI), it will be converted to Unicode. If it is Unicode, it will be converted to ASCII (ANSI).

BMConv

This allows you to take Windows bitmap files, and combine them in one Multiple BitMap file that Symbian OS can understand and use in OPL. BMConv and other issues around graphics are discussed in more detail in Chapter 6.

MakeSIS

This allows you to take your compiled program, the resource files, libraries, and any other files to be packaged up in a standard Symbian Installation System. SIS files are the standard way to distribute Symbian OS applications to phones. Distributing applications is covered in more detail in Chapter 8.

2.3.6 Symbian OS File Manager

Communicators

The Nokia Communicator has a file manager built-in – this can be found under the Office quick launch button, and is called 'File manager'.

Series 60

Series 60 does not come with a file manager as standard, although some phones (e.g. the Siemens SX-1) do ship with a file manager built-in.

There are a number of file managers available for Series 60, and you can use whichever you feel most comfortable with. When discussing Series 60 OPL in this book, we will use FExplorer, which is a freeware file manager available from ***www.gosymbian.com***. Download FExplorer and install the SIS file as you would for any other application – but note this is not supported or endorsed by either me or Symbian.

UIQ

For UIQ phones, there is a free developer tool called QfileMan, which you can use on your own phone. You can get this from the support site

for this book at ***www.symbian.com/books/rmed/rmed-info.html***. Some UIQ models (such as the P900) come with a File Manager built in.

Epocware's PC File Manager

Another alternative is to manage your phone's directories and files using a PC-based file manager. Third-party developer Epocware supply a product called PC File Manager. When your Symbian OS phone is connected via the PC Suite, the PC File Manager opens a window similar to Windows Explorer, and you can drag and drop files onto and around your phone. It is available as a 30-day trial version from ***www.epocware.com***. Again, this is not supported or endorsed by either myself or Symbian.

2.3.7 Creating and Compiling OPL on the Phone

As well as installing the OPL Runtime onto your Symbian OS phone, you can also install a program called TextEd or 'Program'. This is a text editor that is specifically geared towards writing OPL programs on the phone. As well as the normal editing commands, you have a translate function built in, and other useful functionality such as jumping straight to procedures by name.

Once your code has been translated you will be given the opportunity to run it immediately.

You can also use Program on your SDK emulator, it's part of the standard Developer Pack binaries. Some people find this easier than OPLTran. As with programming, there's no 'correct' method – just work in whichever way suits you.

2.4 How we Program

2.4.1 The Development Cycle

Developing programs is a process much like a loop inside a program. You repeat lots of small steps again and again until you get a successful result. That result is the finished program. The steps you go through are:

- edit your source code
- translate the source code
- run the compiled code
- edit your source code if needed to fix any problems
- repeat.

With every program you write, you will go through these steps countless times – as you add new sections to the code. Let's look at each step in turn.

2.4.2 Source Code

Whichever environment you write your code in, the code you write and the techniques used will be the same.

Let's open a new .tpl file in your preferred text editor (or if you're using Program on your Symbian OS phone, a new .opl file). Type in the following lines exactly as they are printed here:

```
PROC Main:
     PRINT "Hello World"
     GET
ENDP
```

Congratulations. You've typed in your first OPL program! Make sure that all the punctuation, spelling, and capitalization are exactly as listed here. Save the file as `HelloWorld.tpl/HelloWorld.opl` before moving to the next step.

Before we move on, a quick word on capitalization. Unlike other computer languages, OPL is not strict on capitalizing procedure, variable, or command names. So `PROC Main:` is the same as `proc MAIN:` or `pROc MaiN:`, but there are certain conventions used in OPL programs that we will use in this book:

- procedure and variable names have the first letter of each word capitalized
- where procedure and variable names contain more than one word, each word is capitalized, but no spaces are included (e.g. `ThisIs-MoreThanOneWord$`)
- commands are fully capitalized (with three exceptions – Graphics commands are prefixed with a lowercase 'g', Menu commands with a lowercase 'm', and Dialog commands with a lowercase 'd' – for example, `gPRINT`, `mCARD`, or `dTEXT`) .

Elements of Hello World

Let's have a look at what makes up our program, line by line.

```
PROC Main:
```

The command `PROC`, short for procedure, gives the next small lump of code its own unique name, in this case "Main". The colon ":" after the

name is always required to signify where the name ends and the code begins.

```
PRINT "Hello World"
```

When an OPL program is first launched, you have a blank screen. In fact, OPL creates a default window that takes up the whole screen – to you it appears that you have a white screen to work with. This is called a window, and we'll look at using windows in Chapter 5. For the moment, we'll just use the `PRINT` command. This takes a string and displays it on the screen.

You should know from the previous chapter that a string is a collection of letters and numbers inside quotation marks. So our `PRINT` command displays Hello World on our empty window, not "Hello World" – if you wanted to display "Hello World" with the quotation marks, your `PRINT` command would read `PRINT """Hello World"""`, with three quotation marks on either side. When your program is run, OPL will know, on seeing three quotation marks, that it should display one on the screen.

```
GET
```

This will get the next keypress event from your phone. An event is something that passes information to the program. This could be a key press, a pen tapping on the screen, a message from the processor, or a variety of other things. Events are how users communicate with your program. In the next chapter, we'll see how events can be read, stored, and used by the program.

`GET` waits for an keypress event to happen. This event is not needed by the Hello World, so we don't read it or use it in any way. But it does mean the text stays on the screen until we press a key.

```
ENDP
```

Signifies the end of our procedure. As it is the first procedure in the program, the program will now stop. If it was the second or subsequent procedure, the program would jump back to the line after that which called this procedure.

2.4.3 Translation

Once you've typed in your OPL source code, you need to translate the source code into object code so the OPL Runtime (on the phone) can understand it. You can do this from inside Program if you are coding on your Symbian OS phone, or through the command line if you are coding on your PC.

Translation from Program (on the Smartphone)

The 'Translate' option can be found under the 'Build' menu. When you press this, your code will be compiled, and the resulting .opo file will be saved in the same directory as the source code.

You will also be given the option to immediately run the .opo code after translation without having to run a file manager.

Translation from PC Command Line

As mentioned previously, OPLTran is a command line tool you can use to compile OPL source code on your PC. Open up your command prompt, and navigate to the directory with the Hello World source code. If you are using the directory structure we've suggested, you should find it at `C:\OPL\HelloWorld\Source\HelloWorld.tpl`. Type in the command line:

```
OPLTRAN HelloWorld.tpl
```

This will compile the code into a file called `HelloWorld.opo`. This file should be copied from your PC to your Symbian OS phone.

2.4.4 Transfer to Phone

This is only needed if you translated your OPL program on a PC. You need to get the .opo file from your PC onto your phone to test it. If you have Bluetooth or an infrared port on your PC you can transfer the file directly to your phone where it should appear in the Inbox and you can try to run it directly from there. Alternatively, you can use a PC-based file manager tool, as mentioned above to copy the .opo file to a location of your choice on your phone and then use a phone-based file manager to run it.

You might want to look at an application called "Forward" (***www.compsoc.man.ac.uk/~ashley***) which allows you to copy a file from the Inbox of a Series 60 device into a specified directory on the C:\ drive of your Series 60 device.

If this all sounds rather convoluted, bear in mind it's only you (the developer) who needs to do this. For any applications you seek to deliver to end-users, you build a SIS file and avoid these problems – more in Chapter 8.

2.4.5 Running Inside the Emulator

If you are working on a PC, then it is possible to run .opo files inside the Emulator that comes with the SDK. You should locate the 'root directory' of the file system for the emulator. If you are using the UIQ

SDK and have installed it to the default path, then you can find it here:

```
C:\Symbian\UIQ_21\epoc32\wins\c\
```

Other SDKs will require you to change the UIQ_21 directory to something more appropriate.

Copy your .opo file to a directory under here. I suggest you create an OPL directory and mirror the directory structure suggested above. Run the file manager inside the Emulator, go to this directory, and run the .opo.

2.4.6 Problems?

Did the program do exactly what you wanted it to? In a simple program like Hello World, probably. But as your programs gain complexity, you'll find that small errors – so-called 'bugs' – may arise.

Any computer is incredibly strict in how you must use it. Substituting `Name$` for `Name%` in your source code may not stop the program translating in all circumstances, but it will stop it running correctly.

One advantage of translating on the phone with Program is that it will show you directly the line where the error occurred. So look carefully at what you see, double checking the syntax of commands. OPLTran will also give you a text output of the line number in question.

Other errors can be from not declaring variables correctly – in which case you might need to look a few lines back in your code to check what you typed when you set something up. OPL has a set of commands called `DECLARE EXTERNAL` and `EXTERNAL`, which help at translate time by ensuring you're using correct variables and procedure names, etc. It is recommended practice to use these commands, but for the programs we're writing in this introductory book we are dispensing with it for brevity. If you're interested in more details, see the command reference in the Appendix.

The hardest errors to find are when the code is typed in correctly, but you have made a logical error. Remember that your program will do *exactly* what you ask it to do, so you should work through your code line by line, following in your head what you are asking your program to do.

Bug hunting is part and parcel of programming, and any programmer who says otherwise is lying. Being able to think clearly at all stages of programming will cut down bugs.

2.5 Summary

This chapter introduced you to both the tools and the elements of OPL. Like any computer language, there are different parts to OPL and we

looked at all of these, and noted the different file extensions used:

- source code (.tpl as a text file on a PC, or .opl on the Symbian OS powered device)
- header files to make reading code easier (.tph or .oph)
- compiled code (.opo)
- compiled code as an application (.app)
- graphics file (.mbm)
- installation file (.sis).

We then looked at one way of developing with all these tools, and showed you the code/test/debug path that you will continue to use as you develop OPL programs.

Finally, we looked at how to run your program on the SDK emulator or on your Symbian OS phone.

3

Event Core

In this chapter you will learn:

- how we take ideas and turn them into OPL code
- about the Event Core: a skeleton program we use as a basis of any new OPL program
- using Event Core on different Symbian OS platforms
- how to store information when you close a program
- how to read in messages from the processor
- how to read and act on pen taps
- how to read and act on key presses.

3.1 Event Core? What is it Good for?

3.1.1 What is Event Core?

So now that we know the basic building blocks of a program and how to use all our programming tools, it's time to start doing some real-world programming.

In the real world, we need to think about how our OPL program will interact with the user, the rest of the operating system, and the processor. To study this, we will build up a skeleton program that does all the things a good program should do. It won't actually 'do' anything useful for a user, but it will provide a starting point for every other program in this book, and can be used by you for all your own programs too.

This core will react to events sent to it from various I/O devices, hence its name, Event Core. Think of it as the foundations of your program, the structure you can build everything else around.

3.1.2 What Should it Do?

Think of the other programs on your phone and what they do. By that I don't mean their high-level function (i.e. the Contacts application stores names and addresses), but how they react to the system and the user as they open, as they are running, and as they are closed.

So our Event Core program needs to be able to accommodate all these core functions of a responsible program:

- react to messages from the processor (for example, exiting the program when the system asks it to)
- save the state of the program when we exit the program
- load these preferences back in when the program is opened again
- be able to display and react to a menu
- be able to recognize and act on a pen tap
- be able to recognize and act on a key press.

We'll work through these elements in this chapter, and by the end of it you'll have a complete foundation for all your future programs.

3.1.3 Recognizing Different Platforms

At the time of writing, Symbian OS has three major platforms that have a working version of the OPL Runtime. It is possible for your OPL programs to be written so that the object code can be run on any of the three platforms. There are a few things we need to consider while writing the Event Core, so that this multi-platform support will be inherent in all your programs:

	Screen (Full)	Screen (with Furniture)	Touch Screen	Keyboard
UIQ	208×320	208×234	Yes	No
Series 60	176×208	176×144	No	No
Series 80	640×200	640×180	No	Yes

Special Keys

CBA1, CBA2, CBA3, CBA4, Menu, Enter, Clear. See Figures 3.1, 3.2 and 3.3.

Our Event Core will have to be able to cope with these different key layouts so that when you come to write your program using the Event Core, you won't need to worry about them.

Figure 3.1 Series 80

Figure 3.2 Series 60

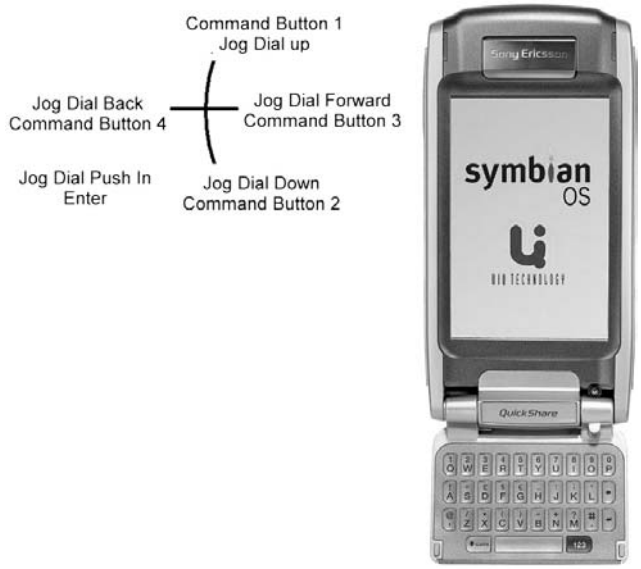


Figure 3.3 UIQ

3.2 Planning the Event Core, Init:

3.2.1 Planning the Event Core

There are some people who can sit down with a blank file and type source code with only a rough idea of what they are planning to do. They see everything, the structure, the variables and all the windows, and can type them straight in.

Most people (including myself) will need to do some planning. We've already bullet pointed out what we want the program to do. We need to break this down into a list of procedures, and work out in what order we move through them. It's also best to add in the basic loops we will use.

So here's our first procedure, in plain English rather than OPL code:

```
PROC Main:
    Set up all the GLOBAL and LOCAL variables
    Load any saved settings
    Check all the things we need to check when starting a program
    Set up program-specific things
    Check for a key press
    Do something if one is pressed
    Keep doing this until we need to exit the program
    Do the things we need to do when exiting a program
ENDP
```

Now some of these lines can be procedure calls, and some can be loops. So let's do a re-write and put this plain English into something very close to OPL:

```
PROC Main:
GLOBAL rem We'll fill these in as we plan the program
LOCAL rem We'll fill these in as we plan the program
    Init:
    InitApp:
    DO
        rem Get a key press (an event)
        rem Act on this key press
    UNTIL rem we need to exit the program
    Exit:
ENDP
```

3.2.2 The Init: (Initialize) Procedure

Again, let's break down what we want to do in this procedure:

- set up paths, determine what disk the program is stored on

- load INI file, determine if this is the first time the program has been run
- sort out screen sizes
- set up windows and load graphics.

We'll look at each of these sections in turn, showing the relevant code, and explaining principles and commands as we come across them.

A Little Reminder

Before all that, we'll just remind the user what program they are running, who wrote it, and what version they are using in a small on-screen message. The command:

```
GIPRINT KAppName$+", "+KAuthorName$+", ver "+KAppVer$
```

is all we need (although note I'll be using `GIPRINT` with a capital 'G,' even if it is a graphics command), and it uses the following constants:

```
CONST KAuthorName$="Ewan Spence"
CONST KAppVer$="1.00"
CONST KAppName$="Event Core"
```

All of these we'll need to define at the start of our program. `GIPRINT` shows an on-screen Graphical Information Printed message for a few seconds, and is great for short bursts of information like this. Its syntax is similar to the `PRINT` statement we used previously.

One good use of `GIPRINT` is that when debugging programs, you can add in a

```
GIPRINT CheckValue$
```

```
GIPRINT NUM$(VariableToShow%,3)
```

. . .to check the value of a variable at that point in a program.

Set Up Paths

Your OPL program is going to need to be able to load information from (and save information to) the phone's storage devices. Graphics are the obvious items here, but no matter what they are, it is important to find out where these stored files are held.

It is a convention of Symbian OS that every program has its own directory to store libraries, graphics, and other information. This directory is:

AnyDrive:\System\Apps\AppName\

In addition, every major program will have this folder on C:\ (the internal disk that is always present) to store its INI file. INI files are only ever stored on the C:\, so we must search for another file to determine where our program is running from. On the assumption that every program will have some graphics, we search for this file to discover what directory the user installed our program and all other supporting files to.

Again, Symbian OS convention is to search for the directory a file is installed on in a specific order. That order starts with the Y:\ drive, then moves back through the alphabet to reach A:\, and finally it checks Z:\ (which is always the 'ROM' of the phone where Symbian OS itself is stored).

While there are no Symbian OS phones with more than one external drive (at the moment), you should always consider the fact that these may get released. Take this into consideration now and you will not need to re-write your code.

So here's what we can do:

```
MbmFile$=KAppNameShort$+".mbm" : Data$="\System\Apps\"+KAppNameShort$
Foo%=ASC("y")
DO
        Drive$=UPPER$(CHR$(Foo%))
        IF EXIST(Drive$+":"+Data$+MbmFile$)
               Path$=Drive$+":"+Data$
               BREAK
        ENDIF
        Foo%=Foo%-1
UNTIL CHR$(Foo%)="Y"
IF Path$=""
       ALERT("Support mbm file not found","Please re-install
           "+KAppName$)
       STOP
ENDIF
```

We have four global variables used here, a constant, and one local variable. This is how they were declared:

```
CONST KAppNameShort$="Core"
GLOBAL Path$(255),Drive$(2),Data$(255),MbmFile$(16)
```

`Data$` will hold the directory where the program stores its information. We use the constant for this short name so when you come to edit the Event Core into your own program, you don't need to search through all the code for the program name, you can just change the constant at the start of the code.

`Drive$` will let us know what drive the information is being stored on (e.g. "C:" – thus it only needs to be two characters long at most).

`Path$` holds the full directory name for the data, and is made up of `Drive$+Path$`.

```
LOCAL Foo%
```

Foo% is used in the DO...UNTIL loop to keep track of the numeric value that represents the letter (of the drive) that we are checking for. When we leave the procedure, the memory space reserved for Foo% is reclaimed as free memory.

Inside the loop, firstly we make sure we are searching for Y: and not y: when we decide what to make Drive$ equal to for this loop iteration, by converting the character into uppercase (using the built-in UPPER$ function). If it is already an uppercase letter (which will only be true in this example when we loop back to Z:) then UPPER$ will have no effect. This exploits how letters are stored on most computers; each letter has a corresponding numerical access code, and capital letters come before lowercase ones. In our code, we start with 'y' and move down a letter until 'a', then roll down one more number which will take us to 'Z', achieving our aim of searching Y: to A: and then Z:.

The EXIST command is a binary check. It returns a 1 if the statement is true, and a 0 if it is false. As you read before, an IF statement is as simple as checking IF something is TRUE, so we can happily write ''if this file exists, then do something''. Which in OPL corresponds to:

```
IF EXIST(Drive$+":"+Data$+MbmFile$)
```

This will be true when we find our .mbm file. When we do, we can set the path and break out of the loop – we don't need to check any further.

```
Path$=Drive$+":"+Data$
BREAK
```

Of course we may loop all the way through the letters (Y: to A:, then Z:, then back to Y:). If this happens, then we can't find the .mbm file, the Path$ will not be given a value, and we must assume the program is not installed correctly. So we show an error message (using the ALERT command) and STOP the program.

```
IF Path$=""
       ALERT("Support mbm file not found","Please re-install
           "+KAppName$)
       STOP
ENDIF
```

The ALERT command is another type of PRINT statement. It puts the information in a box similar to the alert boxes of the Symbian OS phone we are running on. Note that each line of text is separated by a comma, and that we can still add together terms, or just display a given string inside "quotation marks").

The INI File Procedures (LoadIniFile% and SaveIniFile%)

Imagine if you had a program where you could set the screen color to your favorite color. Now whenever you run the program you'd expect the program to remember your choice. This is where the INI file comes in.

The INI file is a very small database that holds information important to the program. If you think of a database as being a large stack of cards, the INI file is represented by one card, which looks a bit like the illustration in Figure 3.4.

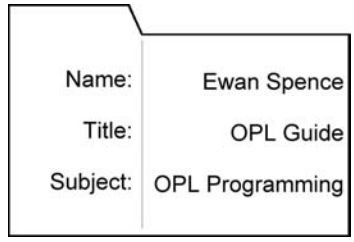


Figure 3.4 Index card with three variables

As with all databases, you have a label (on the left) and the value (on the right). Note how similar this is to assigning a value to a variable. Databases are just big ways to store variables on the disk when a program is not running.

We illustrate the concept of an INI file in Event Core by saving one value – later examples (see Chapter 6) will create much larger databases that we can manipulate, but for now, we'll use the simplest database possible.

So how do we open and read our INI file? As always, think about the steps in English first:

- is there an INI file already? If there is, read the values
- if there isn't, create a set of default values.

In fact, we'll change that last line very slightly to read:

- if there isn't, create a set of default values, then save those values to a new INI file.

So we'll also need a way to save the values when any of them are changed, and when we exit our program. There are two main operations in these statements; reading the values in the INI file; and saving the INI file. Creating the INI file will only ever be done if the reading fails, so is part of that procedure. Let's put this into something a bit like OPL code:

```
PROC LoadIniFile%:
    IF there is an INI file
        Read the values into memory
    ELSE
        Create some default values in memory
        Save these values (call PROC SaveIniFile%: to avoid duplicating
            code)
    ENDIF
ENDP

PROC SaveIniFile%:
    Delete any existing INI file
    Create an empty database card
    Write the values on the database card
    Save the database card to storage
ENDP
```

Note the `%` at the end of the procedure names. This is because when we are finished, we will `RETURN` a value to the procedure that called these two procedures. The `RETURN` keyword allows your procedure to pass back some kind of result to whichever other procedure called it. Much like variables, the type of the result you pass back is specified in the procedure name – so `PROC ReturnsString$:`, `PROC ReturnsLongInt&`, `PROC ReturnsShortInt%` or `PROC ReturnsFloat` are all valid names. You simply then use `RETURN` to pass back the information you want. Calling `RETURN` brings the execution of that procedure to an end, much like as if `ENDP` had been reached. Here, if the INI file exists, we will return a value of 0, if we have to create a new INI file, then we will `RETURN` a value of 1. If we've had to create the INI file, we can assume that this is the first time the program has been run. So in `PROC Init:` we load the INI file with:

```
FirstRun%=LoadIniFile%:
```

This way, `FirstRun%` takes on the value of whatever we `RETURN` in `LoadIniFile%:`.

You've already seen how we look for a file when looking for the .mbm file, so you should recognize the `IF` statement that looks for the INI file:

```
PROC LoadIniFile%:
        IF EXIST ("C:"+Data$+KAppNameShort$+".ini")
               OPEN "C:"+Data$+KAppNameShort$+".ini",A,SoundVol%
               SoundVol%=A.SoundVol%
               CLOSE A
```

```
                RETURN 0
        ELSE
                rem Set initial values here, then save to disk
                SoundVol%=0
                SaveIniFile%:
                RETURN 1
        ENDIF
ENDP
```

So let's look first at what happens if the INI file exists. Well, the first thing to do is `OPEN` the database with the `OPEN` command, which is laid out like this:

```
OPEN Filename, Reference, Variable1, Variable 2, Variable 3, etc.
```

The filename is constructed as before (and again note we use the constant string that holds the name, not the name itself).

It is possible to have more than one database open at a time, so we give each open database a reference letter. In this case, we've used the letter "A". In OPL, you can use the letters A to Z to reference databases.

Finally, we need to know how the database is made up, by listing all the variables that are on the card. In the INI file, we'll only have one variable, but you can have up to 32. Think of these as the heading names of each field of your database.

```
OPEN "C:"+Data$+KAppNameShort$+".ini",A,SoundVol%
```

Thus our INI file has one variable (`SoundVol%`) and is reference letter "A".

We can then read the values stored in the database by using the reference letter, followed by a dot, followed by the name of the field. So we can copy the database value of `SoundVol%` to our `GLOBAL` variable `SoundVol%` with the following:

```
SoundVol%=A.SoundVol%
```

That's us finished with the database! As with anything in programming, when you're finished with something, close it, put it away, or destroy it. Here we `CLOSE` the database with reference "A" and `RETURN` the value 0, leaving the procedure.

```
CLOSE A
RETURN 0
```

What if the INI file doesn't exist? Well, we set a default value to the `GLOBAL` variable `SoundVol%` and call `PROC SaveIniFile%:`

```
SoundVol%=0
SaveIniFile%:
RETURN 1
```

When we come back from PROC `SaveIniFile%:` we `RETURN` the value of 1, representing the fact that we had to create a new INI file.

Now let's think about the `SaveIniFile%:` procedure in English:

- if the directory doesn't exist, create it
- if there is already an INI database with old values, then delete it
- create a new INI database
- save the values to the database
- close the database.

And now in pseudo-OPL:

```
PROC SaveIniFile%:
   Make Directory "C:"+Data$
   Delete the database "C:"+Data$+KAppNameShort$+".ini" if needed
   Create a new database in the same location
   Save the SoundVol% value to the database
   Close the database
ENDP
```

Finally, here it is in OPL:

```
PROC SaveIniFile%:
        TRAP MKDIR "C:"+Data$
        TRAP DELETE "C:"+Data$+KAppNameShort$+".ini"
        TRAP CREATE "C:"+Data$+KAppNameShort$+".ini",A,SoundVol%
        A.SoundVol%=SoundVol%
        APPEND
        CLOSE A
ENDP
```

This introduces us to a new concept – error handling. Normally, when an error happens on running an OPL program, the program will stop and display an error message. The `TRAP` command will trap or suppress any error messages that occur due to a line of code that follows it, allowing the program to continue running. The reason we do this when we `MKDIR` (Make a Directory) is that there is every chance the directory will already be there, but if this is the first time we are running the program, it won't be there. This way, if it is not there, we will make the directory, if not, then the program will raise an error, it will get `TRAP`ped and we can move on and ignore it.

The same reasoning is used for deleting the existing INI file. If there was no INI file and we tried to DELETE it, then we would have an error message. TRAPping again lets us carry on.

The CREATE database command is identical in layout to the OPEN database command. We supply a filename, a reference letter, and a list of variable names.

To assign a variable's value from our program variables to one in the database is simple; we just use the reference letter and the name of the field:

```
A.SoundVol%=SoundVol%
```

However, all we've done is assign values. We haven't actually written them into the database yet. Once we've assigned all the values, we need to create a record (like a paper index card) and add that record into the empty database (e.g. a large empty filing box). This is where the APPEND command comes in:

```
APPEND
```

This adds the record we have just created (with A.SoundVol%=SoundVol%) to the end of the database. Of course, because we've deleted any existing database and created a new database, this appended record will become the first (and only) record.

As before when loading the INI database file, when we're finished we close the database, again with the appropriate close command:

```
CLOSE A
```

Sorting out Screen Sizes

Windows are a bit like pieces of paper on (and off) the screen. If we want to show something on the screen, we need to write it on one of these pieces of paper. We're going to look at windows and graphics in a later chapter, but here are some basics.

When an OPL program is started, it creates a default window. This window fills the screen, so we can use this window to get the size of the screen of the phone we are running on. It is also made the current window; the current (or active) window is the one where we can draw to.

We will store these dimensions in two global variables ScreenWidth% and ScreenHeight% using the following commands:

```
ScreenWidth%=gWIDTH
ScreenHeight%=gHEIGHT
```

Because every Symbian OS phone has a slightly different way of presenting information, we need to realize where all the toolbars and widgets are on screen, so we can calculate the size of the 'empty' window (or 'canvas') we can use.

Here are the variables being defined at the top of our code:

```
GLOBAL ScreenMenubarOffset%, ScreenMainViewWindowOffset%
GLOBAL ScreenStatusBarHeight%, ScreenLeftOffset%
GLOBAL ScreenRightOffset%, Platform%
```

And here is the code to calculate the size of the empty main window. We're also using the variable `Platform%` so we can reference what platform we are running on – this information may be needed at another point in the program.

```
IF ScreenWidth%=640 AND ScreenHeight%=200
        rem Series 80 Communicators
        Platform%=KPlatformSeries80%
        ScreenMenubarOffset%=0
        ScreenMainViewWindowOffset%=20
        ScreenStatusBarHeight%=0
        ScreenLeftOffset%=0
        ScreenRightOffset%=0
ELSEIF ScreenWidth%=176 AND ScreenHeight%=208
        rem Series 60
        Platform%=KPlatformSeries60%
        ScreenMenubarOffset%=0
        ScreenMainViewWindowOffset%=44
        ScreenStatusBarHeight%=20
        ScreenLeftOffset%=0
        ScreenRightOffset%=0
ELSEIF ScreenWidth%=208 AND ScreenHeight%=320
        rem UIQ
        Platform%=KPlatformUIQ%
        ScreenMenubarOffset%=24
        ScreenMainViewWindowOffset%=44
        ScreenStatusBarHeight%=18
        ScreenLeftOffset%=0
        ScreenRightOffset%=0
ELSE
        rem Any new platforms will default to a full screen view
        Platform%=KPlatformGeneric%
        ScreenMenubarOffset%=20
        ScreenMainViewWindowOffset%=0
        ScreenStatusBarHeight%=0
        ScreenLeftOffset%=0
        ScreenRightOffset%=0
ENDIF
```

The values taken by `Platform%` are constants, which are variables we pre-define at the start of our program to make our code easier to read. To make sure this technique works, add the following to the very top of your OPL source code file:

```
CONST KPlatformGeneric%=0
CONST KPlatformSeries80%=1
CONST KPlatformSeries60%=2
CONST KPlatformUIQ%=3
```

We can now calculate our canvas size (i.e. the area we can actually draw to) for this phone:

```
CanvasWidth%=ScreenWidth%-ScreenLeftOffset%-ScreenRightOffset%
CanvasHeight%=ScreenHeight%-ScreenMainViewWindowOffset%-
    ScreenStatusBarHeight%
```

Set up Windows and Load Graphics

There are two types of windows. The first are ones we can display on the screen; these are called drawables. The second are graphics you load into memory from a file; these are called bitmaps. When you create a window, it is assigned a number, and this number is used throughout the rest of the program to refer to it.

These numbers are held in a `GLOBAL` array we create called `Id%()`. So this is another variable we will need to define at the start of our code – as you sketch out your program you should keep a note of all these variables, as it will make it easier when you come to write your own source code.

The `gCREATE` command creates a window that is displayed on the screen. It looks like this:

```
gCREATE(X%, Y%, Width%, Height%, ColorDepth%, Visibility%)
```

`X%` and `Y%` represent the top left corner of the window we are creating. The very top left of the screen on your device is (0,0). You then specify the width (`Width%`) and height (`Height%`) of the window.

`ColorDepth%` is the number of colors that the window can show. The lower the number of colors, the less colors there are! There are values in Const.oph that relate to the value that needs to be used in the `ColorDepth%` variable.

Colours	Value
16	KdefaultWin16ColorMode%
256	KdefaultWin256ColorMode%
4096	KDefaultWin4kMode%
65,535	KdefaultWin64kMode%
16 million	KDefaultWin16MMode%

Finally, `Visibility%` is a binary value, where 1 means the window can be seen, and 0 means it is invisible, no matter what we draw in it.

In Event Core, we have one drawable, defined by this command:

```
    Id%(KMainViewWindow%)=gCREATE(ScreenLeftOffset%,
        ScreenMainViewWindowOffset%,(Ca
nvasWidth%-ScreenLeftOffset%-ScreenRightOffset%),
CanvasWidth%,CanvasHeight%,KDefaultWin4kMode%,1)
```

Now this may look unnecessarily long, but do you remember where we calculated various screen sizes depending on what machine we are using? This uses those numbers so that the same line of code will make sense no matter what platform you run Event Core on.

We'll look at bitmaps and graphics handling in more detail in Chapter 5, so just be aware that this line loads a bitmap from a file into the memory:

```
    Id%(9)=gLOADBIT(Data$+MbmFiles$,0,0)
```

Why 9 and not 2? Well, it's up to you, but when I write code, I assign `Id%(1)` to `Id%(8)` as drawables, and `Id%(9)` and greater as bitmaps. Directly after the `gLOADBIT` command we have three elements. The first is the location of the MBM Graphics files – again note we're using elements that are defined previously, keeping this code as portable as possible. The first 0 is a write protect flag. If it is 0, we cannot edit the graphic (if it is 1, we can). Unless you have a need to change a graphic permanently, you should always leave this as 0.

Finally, the last number lets the command know which bitmap to load. As an .mbm file holds multiple bitmaps (hence MBM), we need to state explicitly which one we're looking at, as a graphics window can only hold one bitmap graphics. The first bitmap in any .mbm file is number 0, followed by number 1, 2, 3, etc.

Putting the Procedures Together

`PROC Init:` has a lot of little things going on that all need to be done every time a program is opened. Here's all the bits put together in the final procedure. Hopefully you can follow this all now:

```
    PROC Init:
    LOCAL FirstRun%,Drive$(1),Foo%
            SETFLAGS &10000             rem Used to allow Auto Switch off
            GIPRINT KAppName$+", "+KAuthorName$+", ver "+KAppVer$
            rem *** Set some names and paths
            MbmFile$="Core.mbm" : Data$="\System\Apps\Core\"
            Foo%=ASC("y")
            DO
                   Drive$=UPPER$(CHR$(Foo%))
                   IF EXIST(Drive$+":"+Data$+MbmFile$)
```

```
                        Path$=Drive$+":"+Data$
                        BREAK
                ENDIF
                Foo%=Foo%-1
        UNTIL CHR$(Foo%)="Y"
        IF Path$=""
                ALERT("Support mbm file not found","Please re-install
                    "+KAppName$)
                STOP
        ENDIF

    rem *** INI File Handling
    FirstRun%=LoadIniFile%:

    rem *** Screen dimensions, color fixed across all devices
    ScreenWidth%=gWIDTH : ScreenHeight%=gHEIGHT
    IF ScreenWidth%=640 AND ScreenHeight%=200
            rem Series 80 Communicators
            Platform%=KPlatformSeries80%
            ScreenMenubarOffset%=0
            ScreenMainViewWindowOffset%=20
            ScreenStatusBarHeight%=0
            ScreenLeftOffset%=0
            ScreenRightOffset%=0
    ELSEIF ScreenWidth%=176 AND ScreenHeight%=208
            rem Series 60
            Platform%=KPlatformSeries60%
            ScreenMenubarOffset%=0
            ScreenMainViewWindowOffset%=44
            ScreenStatusBarHeight%=20
            ScreenLeftOffset%=0
            ScreenRightOffset%=0
    ELSEIF ScreenWidth%=208 AND ScreenHeight%=320
            rem UIQ
            Platform%=KPlatformUIQ%
            ScreenMenubarOffset%=24
            ScreenMainViewWindowOffset%=44
            ScreenStatusBarHeight%=18
            ScreenLeftOffset%=0
            ScreenRightOffset%=0
    ELSE
            rem Any new platforms will default to a full screen view
            Platform%=KPlatformGeneric%
            ScreenMenubarOffset%=20
            ScreenMainViewWindowOffset%=0
            ScreenStatusBarHeight%=0
            ScreenLeftOffset%=0
            ScreenRightOffset%=0
    ENDIF
    CanvasWidth%=ScreenWidth%-ScreenLeftOffset%-ScreenRightOffset%
    CanvasHeight%=ScreenHeight%-ScreenMainViewWindowOffset%-
        ScreenStatusBarHeight%

    rem *** Create initial windows
    Id%(9)=gLOADBIT(Data$+MbmFiles$,0,0)
    Id%(KMainViewWindow%)=gCREATE(0,KMainViewWindowOffset%,CanvasWidth%,
        CanvasHeight%,1,KDefaultWin4kMode%)
ENDP
```

3.3 Other Procedures

3.3.1 The InitApp: (Initialize this Application) Procedure

`PROC Init:` is a generic routine. It will very rarely change when you start a new program. Apart from the layout of windows and graphics that you will load in, it will stay static. The second initialize routine, `PROC InitApp:`, will be very program-dependent, as it covers anything that needs to be set up for your specific program.

In Event Core, we have a small `InitApp:` that simply displays one graphic on the screen and some text. As we'll see, other programs can get quite involved in the `InitApp:`, so again it needs careful planning. Here's the plain English version of our Event Core procedure:

```
PROC InitApp:
    Start using to the main drawable window
    Show the graphic we've loaded
    Print some text
ENDP
```

For this procedure, let's look straight at the OPL code:

```
PROC InitApp:
        gUSE Id%(KMainViewWindow%)

        gFONT KOplFontLatinPlain12&
        gAT 0,CanvasHeight%/2
        gPRINTB "OPL Event Core",CanvasWidth%,3

        gAT 0,(CanvasHeight%-20)
        IF Platform%=KPlatformGeneric% : gPRINTB "Generic
            Machine",CanvasWidth%,3
        ELSEIF Platform%=KPlatformSeries80% : gPRINTB "Series 80
            Communicator",CanvasWidth%,3
        ELSEIF Platform%=KPlatformSeries60% : gPRINTB "Series 60
            Device",CanvasWidth%,3
        ELSEIF Platform%=KPlatformUIQ% : gPRINTB "UIQ
            Device",CanvasWidth%,3
        ENDIF

        rem *** Displaying an MBM (loaded in PROC Init:)
        gAT (CanvasWidth%-127)/2,(CanvasHeight%-65)
        gCOPY Id%(9),0,0,127,41,3
ENDP
```

We can have more than one drawable window open, and only one can be active at a time. It is important to make sure that we're using the right one. The `gUSE` command takes the values that we used when using the `gCREATE` command to identify the windows, in this case the constant representing the Main Window:

```
gUSE Id%(KMainViewWindow%)
```

The numbers in the gCOPY statement tell OPL which part of a graphical bitmap to copy to the screen. We'll discuss these numbers and bitmaps in more detail in Chapter 5. Next are three new commands that give you much more control when printing text onto the screen.

gFONT allows you to change the font of any text that is printed after this command is read. Values for the available fonts are held in the Const.oph file, so you can use these names rather than the numbers:

Font	Value
Latin	KOplFontLatinPlain12&
Latin	KOplFontLatinBold12&
Latin	KOplFontLatinItalic12&
etc.	etc.

In any editing application (such as the text editor you are using) there is a flashing cursor to show where the next character will appear. OPL has two cursors. One for text using the PRINT commands, and one graphical cursor. gAT affects the graphical cursor operations. It starts at (0,0) when you first create a window. You move it by simply stating where you would like it to go with:

```
gAT (X%,Y%)
```

where (0,0) is the top left corner of the current window. In our PROC InitApp: for Event Core we again use the numbers worked out previously for screen dimensions. This is why they were created as GLOBAL variables, so they could be used in different procedures.

gPRINTB is short for Graphical Print Banner:

```
gPRINTB Text$,Width%,Alignment%
```

It takes the cursor as the bottom left of the banner strip it will print the contents of Text$ in. This banner has a specified width (Width%) and can be left, center or right justified using the Alignment% variable: see Figure 3.5.

Alignment	Value	Constant
Right	1	KDTextRight%
Left	2	KDTextLeft%
Center	3	KDTextCenter%

So in our Event Core example:

```
gPRINTB "OPL Event Core",CanvasWidth%,KDTextCenter%
```

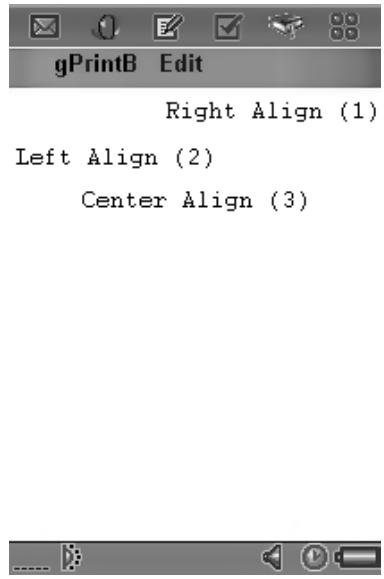


Figure 3.5 gPrintB spaces and line up

we are printing the text "OPL Event Core" centrally inside our banner, which is the width of our main screen.

The `IF` statement has a colon right after it, then another OPL command. Why are these not over two lines? Well, there's no reason to not have them over two lines. The following is also correct:

```
IF Platform%=KPlatformGeneric%
       gPRINTB "Generic Machine",CanvasWidth%,3
ELSEIF Platform%=KPlatformSeries80%
       gPRINTB "Series 80 Communicator",CanvasWidth%,3
etc....
```

But there are times when it is easier to lay out code with more than one statement on a line. You can concatenate statements using a colon, with a space on either side of it. So this is also correct:

```
IF Platform%=KPlatformGeneric% : gPRINTB "Generic
    Machine",CanvasWidth%,3
ELSEIF Platform%=KPlatformSeries80% : gPRINTB "Series 80
    Communicator",CanvasWidth%,3
etc....
```

3.3.2 The Main DO...UNTIL Loop

Let's look back at our pseudo-OPL for the main procedure:

```
PROC Main:
   GLOBAL rem We'll fill these in as we plan the program
```

```
    LOCAL rem We'll fill these in as we plan the program
    Init:
    InitApp:
    DO
        rem Get a key press (an event)
        rem Act on this key press
    UNTIL rem we need to exit the program
    Exit:
ENDP
```

Now we have everything set up, variables loaded, and the main screen showing what we want it to show, we can start doing the central loop of our program. The `DO...UNTIL` loop will have the program iterate through the 'get an event, act on event' cycle until we want to exit the program. As the program reacts to those events, things will happen.

Before we look at those events, let's look at how we get out of the loop first. For this, we'll need to set up some error trapping.

Error Trapping in OPL

We're going to base this loop (and exiting the loop) around error trapping. We've already seen (when loading the INI database file) how to `TRAP` an error so that a program does not display an error message and stop. What we are going to do now is give the program specific instructions on what to do if there is an error, rather than just simply stop.

Firstly, we'll need to know when an error has occurred. We do this by creating a `LOCAL` variable (we'll use `E` in Event Core) and letting it equal the last error OPL knows about (normally zero for no error).

Next, what to do when an error occurs? This is where the `ONERR` command comes in. After using `ONERR` we must tell the program to jump somewhere *in the procedure that the ONERR command is*. How do we do this? We use the `GOTO` command.

`GOTO` immediately jumps the flow of the program from where it is to another line that is labeled with a given name. It is especially useful in the error handling routine as it forces the program to go where we need it, no matter where it is when the error happens.

So at the top of our `PROC Main:` let's set up this error handling mechanism:

```
LOCAL E
E=ERR : ONERR OFF
ONERR Error::
```

To be safe, we switch all the existing error trapping off (using `ONERR OFF`) before we set up our own error handler. In this case, when there is an error, we'll jump to a label called `Error::` [note that all labels end with a double colon (::)].

So after all our code in `PROC Main:` we need to add the following code:

```
Error::
rem *** Show what the error is
dINIT "Error"
dTEXT "",ERRX$,KDTextCenter%
dTEXT "",ERR$(ERR),KDTextCenter%
dBUTTONS "Done",KdBUTTONEnter%
LOCK ON :DIALOG :LOCK OFF
GOTO Top::
```

This brings in one of the main OPL 'widgets'. Dialogs (and Menus) will be looked at in depth in Chapter 4; suffice to say this code will display a box explaining the error number and a short description of what the error is.

But what about the `GOTO Top::` label? This actually takes us back into our main loop so the user can carry on using the program, but hopefully reporting the error back to you, the programmer.

So with our error code, our pseudo-OPL looks like this:

```
PROC Main:
   GLOBAL rem We'll fill these in as we plan the program
   LOCAL rem We'll fill these in as we plan the program
   ONERR Error::
   Init:
   InitApp:
   Top::
   DO
      rem Get an event
      rem Act on this event
   UNTIL rem we need to exit the program
   Exit:
   Error::
   PRINT message and carry on
   GOTO Top::
ENDP
```

Back to Breaking out of the Loop

So one way we can come out of the loop is this:

```
UNTIL (ERR<>E)
```

where `ERR` is the value of the last error, or 0 if there is no error. So when we have an error, `ERR` and `E` will not be equal (i.e. `ERR<>E` will be true).

Doing this ensures that all errors will be passed to the error handler. Of course, when an error happens we will be returned to the program, so we need another way to force the program to stop.

We create a GLOBAL variable called Breakout%. When all variables are created, they initially hold the value of 0. So when we want to exit, we set the Breakout% flag to something other than 0. Which gives us this:

```
UNTIL (ERR<>E) OR (Breakout%<>0)
IF Breakout%
        Breakout%=0
        Exit:
ENDIF
GOTO Top::
```

We reset the value of Breakout% once we have exited the loop as there is a chance we could be returned from PROC Exit:. This is good defensive programming. Even though you are sure you won't be back here, just in case for some unknown, unforeseen reason you *are* sent back here, in this case the Breakout% variable will still 'behave' itself.

Note also that if we come out of the DO...UNTIL loop because of the ERR value, we won't automatically reach the jump to PROC Exit:, rather we'll find the GOTO Top:: label and be returned to the DO...UNTIL loop, even if something goes wrong with the error handler. Again, good defensive programming!

3.3.3 PROC Exit:

When we're ready to exit our program, there are a few things we need to do. These will be handled in PROC Exit:

```
PROC Exit:
LOCAL Foo%
       ONERR JustStop::
       SaveIniFile%:
       Foo%=0
       DO
              Foo%=Foo%+1
              IF Id%(Foo%)
                     gCLOSE Id%(Foo%)
              ENDIF
       UNTIL Foo%=KMaxWindows%
       JustStop::
       ONERR OFF
       STOP
ENDP
```

We set up a new error handling procedure here. If something goes wrong in the Exit: procedure, rather than report the error and try to go back to

the program, we're going to ask the program to just stop anyway. `ONERR JustStop::` takes care of this.

Next we save the INI database file by calling `PROC SaveIniFile%:`. We discussed this procedure earlier, and although there it was used to save the default settings, here it will be used to save whatever the settings are currently.

Now we set up a small `DO...UNTIL` loop that closes all the drawable windows and graphics that we have loaded. In theory, the OPL Runtime should recover the memory used by these elements when we close the program, but there is nothing wrong with giving it a helping hand. A temporary `LOCAL` variable (`Foo%` again) is used to count up from 1 to the maximum number of windows (something that we set as a constant at the start of the source code) and perform the `gCLOSE` command on each element in turn.

Finally, we give the `STOP` command, which stops the OPL program running.

3.3.4 Getting Key Presses, Events, and Commands

So now we have everything in `PROC Main:`, except how to actually receive any inputs (events) from the device or the user.

We need to react to three types of events:

- system commands (some of these are defined in Const.oph such as `KGetCmdLetterExit$, KGetCmbLetterBroughtToForeground%, KGetCmdLetterBackup$`)
- pen taps
- a key being pressed.

Note that not every Symbian OS phone will have all of these elements (for example, current Series 60 phones do not have a touch screen), but to make our Event Core portable we can address them all anyway.

System Commands

When the processor wants your OPL program to do something, it will send a command 'word' to the program. This word will be stored by the runtime until you decide to read it into a string variable (of maximum length 255 characters) with the following command:

```
Command$=GETCMD$
```

The first letter reflects the action that needs to be carried out. This letter value is also stored as a constant in Const.oph – as always, using the values in Const.oph will make your program easier to read and to port to other versions of OPL.

The rest of the string will be a filename. Most of the time this will be to the program file, but if you create an OPL program that uses files (e.g. a text editor) it will refer to the open file. Note that file-based programs are not discussed in this book.

Letter Sent	Constant Value	Requested Action
X	KGetCmdLetterExit$	Close your program immediately
?	KGetCmdLetterBackup$	Phone is being backed up, close program
	KGetCmdLetterBrought-ToFGround$	Program brought to foreground
		Program sent to background
O		Open a specified file
N		Create a new blank file

All good programs should react to system events. In Event Core we check the command word before checking for any events with this command, and take the appropriate action.

How Do you Read an Event?

First we define a `GLOBAL` array that will be used to store any event we read. When we want to read an event, we use the `GETEVENT` command to read in all the relevant info into the 32-bit array previously defined:

```
GLOBAL Event&(16)
...
GETEVENT32 Event&()
```

When an event occurs, the details of the event are stored in a queue. The `GETEVENT32` command will take the first event in the queue, and then move to the other events. So if you have two events happening very close to each other, the first will be read, and the second will wait until `GETEVENT32` is used again.

`GETEVENT32` will fill the 16 elements of its array with the following information. Not all of these are used, so only those relevant to OPL are listed:

Event&()	Value	Type
1	Unicode Value	Key pressed
2	Timestamp	When the key was pressed
3	Window Number	Which window the pointer event is in
4	1	Pointer removed from screen (pen up)
5	0	Pointer tapped onto screen (pen down)
6	Integer	X Coordinate of pen tap
7	Integer	Y Coordinate of pen tap
8	Integer	X Coordinate of pen tap relative to parent window
9	Integer	Y Coordinate of pen tap relative to parent window

We'll now check the `Event&` array for our two main types of event.

PROC PointerDriver:

The pointer can be used in two main areas. The first is to activate the menu on UIQ devices. The second is to register a tap on the screen.

If you tap the menu bar in an OPL program on UIQ, it will pass an event representing this menu press as if you had pressed a key. This value is `KMenuSilkScreen%`, and is picked up in our key event handler. Here we will discuss pen taps that are in the working area of the screen.

There are three types of pen events, and they all report back three numbers, via the `GETEVENT32` command (see above table):

- the graphics window the event occurred in (remember the `Id%` window array? It's these numbers that are reported)
- the X coordinate in the window the event was in
- the Y coordinate in the window the event was in.

In the case of the Symbian OS phone not having a touch screen (such as a Series 60 or Communicator), it is not possible for a pointer event to be passed to the program. However, the Pointer Driver code can happily stay inside the program and simply never get called.

PROC KeyboardDriver:

We split the types of keyboard events into two areas. The first is direct key presses such as pressing cursor keys, the enter key, buttons that select options, etc. You'd use these direct keys in a game, or when scrolling through a list.

The second type of keyboard event is a hot key – this is commonly used in devices such as the Communicators, but has less relevance on UIQ devices.

After a key is pressed, and it is not one of the 'direct' keys (for example, the cursor keys could be direct keys used to choose an item from a list), we look at what modifier keys are used. If there are any present, this indicates a hot key has been pressed by the user – these are keyboard shortcuts to various menu items or functions. In the case of Communicators, for example, hot keys use a combination of Ctrl and Shift keys alongside normal keyboard letters.

Ctrl-K is a popular hot key. This brings up any settings or preferences dialog the program has. Note that Event Core does not read the Ctrl key as a distinct key press, but notes the fact that it has been pressed. The same applies to the Shift key. In our code, we have two variables used to track these:

```
Mod%=Ev&(4) AND 255
IF Mod% AND 2 : Shift%=1  : ELSE : Shift%=0   : ENDIF
IF Mod% AND 4 : Control%=1 : ELSE : Control%=0 : ENDIF

rem Check for Direct keys here
IF Shift% : Key&=Key&-32 : ENDIF

rem Check for Hot Keys here
IF Key&<=300
       ActionHotKey:
ENDIF

rem reset Modifiers
Control%=0 : Shift%=0
```

Where do we get these values for `Key&`? There is a list of character codes for all the keys in the SDK documentation. For example, 32 is the code for the space bar and 13 is the Enter key.

We can have as many parts to the `IF` statement that checks for the space bar (and we'll see this in later programs), but once these run out, we check to see if a hot key has been pressed, by jumping to `PROC ActionHotKey:`.

Processing Hot Keys

A separate procedure for hot keys is used to work out what procedure to jump to when a hot key is pressed. This is because the menu system also uses the hot key system, but we'll come back to that in the next chapter.

Now you could do a simple `IF Key%=48` (where 48 is the character code for the letter "a"), but the `ASC` function gives the character code for any requested character. Therefore we can take the numerical value from `Key&` and compare it to `ASC(Value$)` of the hot key to check the result, and make our code more readable:

```
PROC ActionHotKey:
     IF Key&=ASC("e")-96 : Exit:
     ELSEIF Key&=ASC("k")-96 : SetPreferences:
     ELSEIF Key&=ASC("A")-96 : About:
     ENDIF
ENDP
```

Once we find the correct hot key, we go off to a procedure, do something, and come back. Depending on what you want your program to do, Event Core's layout and this 'jumping' method should see you through most exercises and tasks for many years to come.

Menu System

Chapter 4 looks at the menu system in more depth, and we will discuss the menu commands and how to use them in Chapter 4.

3.4 Summary

This chapter introduced you to the Event Core, a framework that gives you something to build your OPL programs into. You've been taken through a huge number of OPL commands, what they do, and how they relate to each other.

If you've got this far and have followed everything, then congratulations. If you're still unsure, re-read this chapter and experiment. Once you understand the principles in this chapter, the rest of the book will make a lot more sense.

We finally looked at reading input through pen taps on the screen, and direct presses of the keys. Chapter 4 will introduce you to the main user elements of your operating system, the Menu bar and Dialog boxes. This will allow you to quickly and efficiently get information and content from the user, as well as present options and information to them.

4

A Conversion Program: Event Core in Practice

In this chapter you will learn:

- how to turn Event Core into a practical program
- creating and using a Menu system
- creating and using Dialog boxes
- working with variables, doing calculations, and displaying the results.

4.1 First Steps with Event Core

Now that we have the Event Core framework, we can use this to create our first real program – one that does more than doing nothing (although Event Core does this so well)!

4.1.1 The Conversion Program

Our first program will be a conversion tool. It will take in measurements of one type, and convert them into another type. We will use the four conversions listed below, and show you how to add in your own conversions to replace or augment these:

- temperature: between degrees Celsius and degrees Fahrenheit
- long distance: between miles and kilometers
- short distance: between centimeters and inches
- weight: between pounds and kilograms.

4.1.2 Starting the New Project

Take the Event Core source code and copy it into a new folder for your project. If you're using the conventions of the book, then create a folder

called `Convert`, and in that folder create a folder called `Source` (for your source code). Rename the Core.tpl to Convert.tpl (or Core.opl to Convert.tpl if you are working directly on the Symbian OS phone).

Opening Convert.tpl you'll want to change the first few constants to reflect the new program:

```
CONST KAuthorEmail$="ewan@izzyhack.org"
CONST KAuthorName$="Ewan Spence"
CONST KAppName$="Convert"
CONST KAppVer$="1.00"
```

I always like to translate the code now and see the new empty program display its own name – it's a bit like conception. Now we have to help this program grow and reach its potential. And to do that, we need a plan.

4.1.3 Planning Convert

We've already defined in bullet points what we want the program to do (the four conversion criteria listed above). We need to think what inputs we're going to need, and what outputs we'll use. What I do here is forget about any limitations of OPL and sketch out on paper what the program would be expected to do, no matter which language it was written in.

We would need a menu system that would let us choose the conversion we want to perform. It would then display a box asking for the initial information (how many miles, in this case let's say 500). The user would then see the result in a second dialog box (which would be 804.67 kilometers).

Now let's think how that could relate to Event Core. As you remember, Event Core loops around waiting for something to happen. In Convert, we're going to wait until the menu is called. When this happens, we'll display all the available conversions, and if one is selected we'll move onto the dialog box to get the relevant figures.

Once this dialog box is completed, we'll show the results in a second dialog box, and then close the dialog box and the menu, returning to the state where we wait for something to happen.

4.1.4 Recognizing and Acting on Menus in OPL Code

The commands to define a menu are as follows:

```
mINIT
mCARD "Convert","Option 1",%a,"Option 2",%b
MenuResult%=MENU(LastMenu%)
```

Starting the Menu

`mINIT` is an initialization command, and lets the runtime know that a menu is about to be created and called. It must always be the first command when you call a menu.

A Single Pane Menu

The `mCARD` command defines a single menu card. Second and subsequent menu cards can be defined by having additional `mCARD` commands.

The first part of the `mCARD` command is the name of the menu card. This is the name that appears in the top menu bar of your program. This name does not appear in a menu on Series 60. You then have as many paired commands as you need to define your menu. The first text string will be the text shown on the menu, and the second value is the integer that will be returned by the `MENU` command. For obvious reasons, this number must not be repeated in any of the `mCARD`s defined for this instance of the menu system.

You could have any number here, but we're using a value of %a for 'Option 1'. This represents the value of the letter 'a' if pressed. If we were to use %b, it would be 'b' and %B would be 'B' (so %b is not the same as %B). %a actually means ''the character code for the letter 'a' which in this case is 97''. These values mean that the relevant menu option will be labeled with the relevant hot key on compatible devices (e.g. Series 80). So %a would be shown with 'Ctrl+A' as the hot key, and %B would be shown with 'Shift+Ctrl+B'.

These returned values will be used to jump to the correct procedure when we act on the command received by the menu. More on this in a minute.

Two or More Menu Panes

Let's say we want a second menu card, so our top menu bar reads ''Convert'' and ''Edit''. To add in a second card, we simply list it after the first card, ensuring the name of the card is unique, and that we do not repeat any of the hot key numbers:

```
mINIT
mCARD "Convert","Option 1",%a,"Option 2",%b
mCARD "Edit","Option 3",%c,"Option 4",%d
MenuResult%=MENU(LastMenu%)
```

You can add in as many menu cards as you feel the need, but there will come a point where your menu bar is too crowded to make any sense.

In all cases, `LastMenu%` should be a `GLOBAL` variable into which the last selected menu item will be stored. This ensures that (on compatible devices) the menu can be brought up offering the last-selected item as the initial choice for the user if required.

Reading the Result

Now we have defined all the elements of a menu, we will want to display it on the screen and read the result. We do this with the `MENU` command

```
MenuResult%=MENU(LastMenu%)
```

This displays the menu system, and once we choose a menu option, it will return the value that is listed after that menu option (the hot key value).

Calling the Menu with a Key Press

Both Series 60 and Series 80 call up menus with a key press. Series 80 has a dedicated key (`KKeyMenu32&`), while Series 60 will generally use the CBA1 button (the left soft key). When the Event Core picks up this value, it will call the menu procedure.

Calling the Menu with the Touchscreen

In a similar way to the Series 60 and Series 80 devices, the UIQ menu call is represented by a constant (in this case `KMenuSilkScreen&`). When the menu area is tapped, it acts as if it was a keypress sending the `KMenuSilkScreen&` value to the Event Core, which is handled in exactly the same way as any other keypress calling for the menu to be displayed.

Hot Keys

We've stressed that we use hot key values in the menu system. We also have a separate routine in Event Core that reads in hot keys from the keyboard (if present).

As you know, Symbian OS applications can often achieve the same result in many different ways. One example of this is choosing a menu option. For example, the option to show an About box can be called by a hot key combination on the keyboard or by selecting the menu item itself. This is how we implement menu functions – by ensuring that the hot keys in our menu system (even if the key combination is not shown on our target device) can re-use the `PROC ActionKey:` procedure to process the menu input.

Rather than read the value returned by `MENU` into an arbitrary variable, we can re-use the `Key&` variable:

```
Key&=MENU
```

and modify PROC KeyboardDriver: by adding the following four lines:

```
rem Call Menu
IF Key&=KKeyMenu32& OR Key&=KMenuSilkScreen&
     DisplayMenu:
ENDIF

rem Check for Hot Keys here
IF Key&<=300
       ActionKey:
ENDIF
```

So when we choose a menu option, through a pen interface, or using the cursor keys and a select button, the value is passed back to PROC ActionHotKey: as if a hot key was pressed (even if the device does not have a keyboard). PROC ActionHotKey: can now jump to the correct procedure, carry out the operation, and then return to the main loop of the program.

4.1.5 Building a Menu System in OPL Code

Cascading Menus

As well as separate menu cards, there is another way to present information in a menu. This is through the use of cascading menus. A cascade in this respect is a menu option that, when selected, will pop up a second list of menu options the user can choose from.

To use this feature, you firstly define the cascading menu by using the mCASC command, in the same way as you would use the mCARD system. The title (first string) in mCASC must be the same text as the menu option that will call up the cascade. Now, as long as the cascade is defined **before** the menu card it will appear in, you can place it in the menu card by using the title of the cascade (as defined at the start of the mCASC command) and appending a ">" symbol. For any menu item that you don't want to have a hot key you can specify a value less than or equal to 32 instead. This always applies to mCASCs. Here, we're using the value 16.

Here's our menu code, now with a cascading menu option called "More" that appears as the last entry in the first menu card:

```
mINIT
mCASC "More","Option 5",%e,"Option 6",%f
mCARD "Convert","Option 1",%a,"Option 2",%b,"More>",16
mCARD "Edit","Option 3",%c,"Option 4",%d
MenuResult%=MENU(LastMenu&)
```

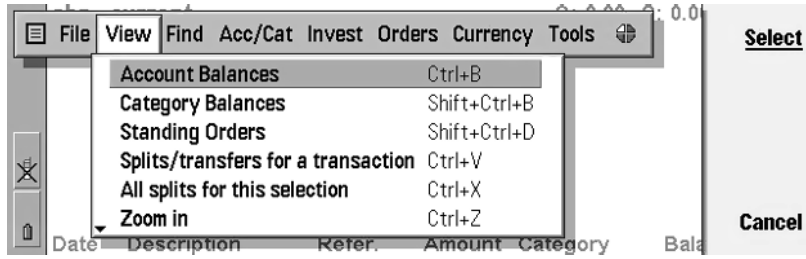


Figure 4.1 ABP menu screen

Series 80 Menus

Series 80 is the closest to a PC or Mac menu on Symbian OS. You can have as many menu cards as can be accommodated on the screen, and while menu cascades can be used, there should be enough room on the cards for all your menu options. See Figure 4.1.

Menus are called up by pressing the Menu key, and this will be picked up in `PROC KeyboardDriver:`. Hot keys are shown, and they should follow the Series 80 style guide. A link to this can be found on ***www.symbian.com/books/rmed/rmed-info.html***.

Series 60 Menus

Series 60 does not have specific menu keys, therefore you should hard code the CBA1 button to call the menu. Generally, Series 60 applications should use the left soft key (CBA1) to call a menu or do an action, and the right soft key (CBA2) to go back/undo an operation or exit the application. When you call a Series 60 menu, you will not see the title of any menu card, and only the first menu card is shown – any others are ignored. You are encouraged to use one level of cascading menus for more options.

Hot key values are not shown on the menu but are still used in the underlying OPL code as the return value for the selected menu item – this means our `ActionHotKey:` procedure will still work. You should always have "Exit" as the last option in a Series 60 menu.

UIQ Menus

UIQ menus can only be called by a pen tap on the screen. This is picked up in the `PROC PointerDriver:` procedure, where the menu `PROC` is called, and the resulting command is passed to `PROC HotKey:`.

UIQ applications generally have two menu cards, one named after the application and one called 'Edit'. Note that the value of constants means our menu card can use the `KAppName$` string constant:

```
mINIT
mCARD KAppName$,"Option 1",%a,"Option 2",%b
mCARD "Edit","Option 3",%c,"Option 4",%d
MenuResult%=MENU(LastMenu%)
```

If you program more than two menu cards, then you will see these menu cards until there is no more room on the screen. Hot keys are not shown, but as with Series 60 they're still used in the underlying OPL code.

UIQ applications do not normally have an exit command, instead they respond to commands from the system to close down when more memory is needed. These messages are handled by the System Command Handler Procedure in our main event loop. The drop down 'views' list on the right-hand side of a UIQ menu bar is not implemented in OPL.

4.1.6 Putting Together Menu Code for Each UI

When programming your menus, you have to consider what UI you will be using. If you know the program will only run on one Symbian OS UI, then you only need to use the relevant menu system.

If the .opo file is designed to run on more than one platform, then you need to define the menu system in such a way that the correct style is used. In `PROC Init:` we set the variable `Platform%` to determine the UI we were using (1=Series 80, 2=Series 60 and 3=UIQ). This gives us something like:

```
mINIT
IF Platform%=KPlatformSeries80%
       mCASC "More","Option 5",%e,"Option 6",%f
       mCARD "Convert","Option 1",%a,"Option 2",%b,"More>",16
       mCARD "Edit","Option 3",%c,"Option 4",%d
ELSEIF Platform%=KPlatformSeries60%
       mCARD KAppName$,"Option 1",%a,"Option 2",%b,"Option
           3",%c,"Option 4",%d
ELSE
       mCARD KAppName$,"Option 1",%a,"Option 2",%b
       mCARD "Edit","Option 3",%c,"Option 4",%d
ENDIF
MenuResult%=MENU(LastMenu%)
```

While we do not check for Platform%=KPlatformGeneric% (an unrecognized UI), it will default to using the UIQ menu (Platform%= KPlatformUIQ%) of two menu cards. This is another example of programming with an eye to what can go wrong (i.e. someone running OPL on a device with a new screen size).

4.1.7 The Convert Menu

So, now we know how to set up a menu, let's put one into action for our conversion program.

Sketch it Out

As with all programming, we'll sketch out what we want the program to do, then put this into a structured format before going ahead and coding that section.

There are some menu options that you should always consider first – the defaults, as it were. These are:

- About
- Exit (although this option should not appear in UIQ applications)
- Preferences.

Add to this the four conversion operations we'll want to show:

- temperature
- long distance
- short distance
- weight.

And we have the options we want to appear on our menu system.

Show the Code

Let's look at the two variants of menu code. The first as if we are coding for only one UI – let's look at the UIQ version:

```
mINIT
mCASC "Units","Temperature",%t,"Long distance",%l,"Short
    distance",%d,"Weight",%w
mCARD "Convert","Units>",16
mCARD "Edit","Preferences",%c,"About",%a
MenuResult%=MENU(LastMenu%)
```

While we've used the Menu Cascade option here when it isn't truly needed, there is scope on how to add in new options that will require there to be this cascade in place.

If we were to do a 'multiple UI' version, the code would look like this:

```
mINIT
IF Platform%=KPlatformSeries80%
       rem Series 80
```

```
        mCARD "File","Exit",%e
        mCARD "Convert","Temperature",%t,"Long distance",%l,"Short
            distance",%d,"Weight",%w
        mCARD "Tools","Preferences",%c,"About",%a
ELSEIF Platform%=KPlatformSeries60%
        rem Series 60
        mCASC "Units","Temperature",%t,"Long distance",%l,"Short
            distance",%d,"Weight",%w
        mCARD "Convert","Units>",-16,"Preferences",%c,"About",%e
ELSE
        rem UIQ Menu (also Default)
        mCASC "Units","Temperature",%t,"Long distance",%l,"Short
            distance",%d,"Weight",%w
        mCARD "Convert","Units>",16
        mCARD "Edit","Preferences",%c,"About",%a
ENDIF
MenuResult%=MENU(LastMenu%)
```

4.1.8 Gathering and Presenting Information – Dialogs

We've now seen many ways to get input from a user through the use of menus and key presses (mainly through hot keys). There are two more ways to take input. Directly from a pen tap on the screen, and gathering information from dialogs.

We'll look at pen taps in the next chapter, but for now let's look at dialog boxes.

Why have Dialogs?

Dialogs are designed to show information to the user, and gather information from the user. They are a two-way process. In Convert, we'll use a sequence of dialog boxes that will firstly get the number we want to convert and secondly show the result.

One advantage of dialogs is that as a programmer you do not need to worry about how the dialog box works. You don't need to worry about reading the keyboard, or processing any handwriting recognition. The OPL Runtime will do this for you. This means that when constructing a dialog box you can be confident that it will work on any of the UI versions.

The OPL Runtime will also lay out the dialog box and give it a similar look to other dialog boxes on that UI. This makes dialog boxes a powerful tool for cross-platform programming.

Showing Info – the About Dialog

Let's start off with a simple dialog box – one to show a simple About screen. See Figure 4.2. Here's the OPL code:

```
dINIT "About "+KAppName$"
dTEXT "","by "+KAuthorName$,2 KgPrintBLeftAligned%
dTEXT "","by "+KAuthorEmail$,2 KgPrintBLeftAligned%
LOCK ON :DialogResult%=DIALOG :LOCK OFF
```

Figure 4.2 Convert about dialogue box

Constructing a Dialog

Creating and displaying your dialog is very similar to creating and displaying a menu. You initialize the dialog, build up the elements, and then show the dialog (while reading the result if appropriate). You then add elements to your dialog box, one line of code at a time. These will be added to the dialog box in the order that they appear in the code. In the example above we have two text lines in our dialog box (all Dialog elements are prefixed with a 'd').

Finally, we display the dialog box with the `DIALOG` command. As with the `MENU` command, `DIALOG` will return a value depending on how we exit the dialog box (if you wanted you could use the `DIALOG` command on its own without recording the value, but this is regarded as bad programming practice).

Looking up Commands (dTEXT and DIALOG)

As with all OPL commands, understanding the syntax of a command will give you more than enough knowledge to implement a command not described in any of these chapters. Every OPL command is listed, with syntax and a short description in the Appendix of this book, and that should always be your first port of call for learning new commands.

Dialog commands are probably the easiest to understand, so let's break down the dTEXT command with help from the Appendix. Here's the entry for dTEXT:

```
*****************************************************************************

  dTEXT              Defines text to be displayed in a dialog

  Usage:             dTEXT p$,body$,t%
  or                 dTEXT p$,body$

  Defines a line of text to be displayed in a dialog.

  p$ will be displayed on the left side of the line, and body$ on the
  right side. If you only want to display a single string, use a null
  string ("") for p$, and pass the desired string in body$. It will then
  have the whole width of the dialog to itself. An error is raised if
  body$ is a null string and the text line is not a separator (see below).

  body$ is normally displayed left aligned (although usually in the right
  column). You can override this by specifying t%:

  Alignment of body$ is only supported when p$ is null, with the body
  being left-aligned otherwise. In addition, you can add any or all of the
  following three values to t%, for these effects:

  Specify this item as a text separator. p$ and body$ must both be the
  null string for this to take effect. The separator counts as an item in
  the value returned by DIALOG.

  These constants are supplied in Const.oph.

  See also dEDIT, dINIT.

*****************************************************************************
```

There are two ways we can type the dTEXT command in our source code, dTEXT p$,body$ or dTEXT p$,body$,t%. Both ways are correct, and neither is more correct than the other. From here we can see the three elements that make up dTEXT, the prompt (p$) the body text (body$), and the formatting (the value of t%).

Where you have a variable in a command explanation, this is for guidance and reference only. You can enclose a string in quotation marks, and this will work just as well as a reference to a variable. So dTEXT "Name","Symbian" is valid, as is:

```
Prompt$="Name"
Body$="Symbian"
...
dTEXT Prompt$,Body$
```

You are not limited to the variable names in the command explanation, you can use whatever you need to use in your program.

The text notes that the formatting will default to left-aligned, but the second way of writing the command (`dTEXT p$,body$,t%`) allows you to specify your own formatting. The first option is where to align the text (left, center, or right) and the second allows you to add other formatting features with which you can experiment to get the correct look. There are more details on the formatting options in the Appendix.

Values Returned from a Dialog

So now we've initialized our dialog, and created the text to display (using `dTEXT` for each line of text), we can display it by using the `DIALOG` command. Here's the entry for Dialog from the command index:

```
*************************************************************************

  DIALOG      Presents a dialog

  Usage:              n%=DIALOG

  Presents the dialog prepared by dINIT and commands such as dTEXT and
  dCHOICE. If you complete the dialog by pressing Enter, your settings are
  stored in the variables specified in dLONG, dCHOICE, etc., although you
  can prevent this with dBUTTONS. If you use dBUTTONS when preparing the
  dialog, the keycode that ended the dialog is returned. Otherwise, DIALOG
  returns the line number of the item that was current when Enter was
  pressed. The top item (or the title line, if present) has line number 1.

  If you cancel the dialog by pressing Esc, the variables are not changed
  and KDlgCancel% is returned:

  KDlgCancel%         0              Return value: dialog was cancelled

  See also dINIT.

*************************************************************************
```

What we have here is this entire section in three paragraphs – explaining exactly how dialogs work, how we start them, construct them and finally display (present) them on the screen. It also explains the value that the `DIALOG` command will pass to the integer variable that is used within the line of code that presents the `DIALOG`.

Getting Input from a Dialog (Prefs)

While a dialog can be used to present information, they are even more useful when used to gather information. Whenever you need to have a number, or a string of text, entered by a user, your first thought should be to use a dialog box.

There are four main dialog commands used to get values from the user. These are:

- dCHOICE
- dEDIT
- dLONG
- dFLOAT

For a full description of these commands (and subsequent commands), look in the Appendix for the command list.

dCHOICE

dCHOICE provides you with a drop down list of text strings to choose from. Let's imagine we want to present the user with a choice of two items. For example, in our temperature conversion, we need to know if it is Celsius to Fahrenheit, or vice versa.

So our two options in the drop down box in the dialog are "Celsius to Fahrenheit" and "Fahrenheit to Celsius". We need to store the result of this choice, and we use an integer number. If we choose the first option then the integer will be 1, and if we choose the second it will be 2. Let's store this value as TemperatureDirection%. Remember that we'll need to have initialized this variable with either LOCAL TemperatureDirection% or GLOBAL TemperatureDirection% at some point previously in the code:

```
dINIT "Choose Conversion"
dCHOICE TemperatureDirection%,"Celsius to Fahrenheit,Fahrenheit to
    Celsius"
LOCK ON :Return%=DIALOG :LOCK OFF
```

dEDIT

dEDIT allows the user to type in a string and have it stored in a string variable when the dialog is successfully closed (i.e. the end user doesn't press Esc or a cancel button). dEDIT isn't used in Convert, but it is one of the more important dialog commands. Because the dialog commands are built in to the runtime, you don't need to worry about reading in keystrokes, or T9 input, or character recognition: this will always be done for you. dEDIT is the easiest way to get the user to enter text strings.

dFLOAT and dLONG

In a similar way to dEDIT, these commands allow you to type a number in. dFLOAT is for very large numbers or numbers with decimal points (e.g. LargeNumber, where there is no sign at the end of the variable name). dLONG is used for long integers (e.g. MyNumber&). Both commands require a lower and upper limit on the number, which you can specify, and a prompt (which can simply be an empty string represented by two quotation marks "", if you really want no prompt):

```
    dFLOAT FloatPoint,Prompt$,MinValue,MaxValue
    dLONG Long&,Prompt$,MinValue&,MaxValue&
```

Again, see the command list for a full explanation.

4.1.9 Putting it All Together

So, let's take the Event Core, what we now know about menus and dialogs, and put together the Convert program.

The user initiates a conversion dialog with either the menu entry, or with a hot key press. For both cases, we need to add some lines into our `PROC ActionHotKey:`

```
PROC ActionHotKey:
     IF Key&=ASC("e")-96 : Exit:
     ELSEIF Key&=ASC("k")-96 : SetPreferences:
     ELSEIF Key&=ASC("A")-96 : About:
     ELSEIF Key&=ASC("t")-96 : DialogTemperatureConvert:
     ELSEIF Key&=ASC("l")-96 : DialogDistanceConvert:
     ELSEIF Key&=ASC("d")-96 : DialogShortDistanceConvert:
     ELSEIF Key&=ASC("w")-96 : DialogWeightConvert:
     ENDIF
ENDP
```

So once a conversion is chosen, we're passed over to a procedure to do that conversion.

The Convert Variables

So we've worked out which conversion we're going to do, and we've been sent to the correct procedure to work out two values. The first is what direction the conversion will go in (e.g. miles to kilometers, or kilometers to miles). The second is the value we are converting (e.g. 500 miles). We'll use a dialog box to gather these, and then pass the two variables to a procedure that will calculate the result, and then display it.

Now, we could set up two `GLOBAL` variables to hold both `ConversionDirection%` and `ConversionValue%`, but remember that `GLOBAL` variables are always present and available to a program, using up memory. There is no need to store these values after the calculation is done, so we will create two `LOCAL` variables with the appropriate names, and then pass these to the procedure that does all the calculations:

```
PROC DialogDistanceConvert:
LOCAL ConversionDirection%,ConversionValue,ConversionReturn, Foo%
    rem get the values from the user
    dINIT
    dCHOICE ConversionDirection%,"Direction","Miles to Km,Km to Miles"
```

```
    dFLOAT ConversionValue,"Value",0,1000000
    LOCK ON :Foo%=DIALOG :LOCK OFF
    IF Foo%=0
           RETURN
    ENDIF
    rem Go and do the calculation in another procedure
    IF ConversionDirection%=1
            ConversionReturn=DoConversion:(1.60934,ConversionValue)
    ELSE
            ConversionReturn=DoConversion:(0.62137,ConversionValue)
    ENDIF
    rem Display result here
ENDP
```

So once we've taken the values from the dialog, we pass them to a procedure that will do the conversion of the values passed to it. 'Passing' variables is a great way to send a fixed number to another procedure when you don't want to use a global variable. To pass a variable, you enclose it in brackets after calling a procedure. You can pass more than one by separating them with a comma.

Here are some examples. Note we can pass string variables, actual strings, and all types of number variable. Both of these are valid ways to pass information to a procedure:

```
    CallCustomer:(Name$,IDNumber%,Number$)
    CallCustomer:("Geoff",67,"+441234567890")
```

Doing the Maths

So at the other end, we need to make sure the procedure is expecting to receive the information passed to it. We do this in a similar way to declaring `LOCAL` variables, except we name the passed information by using names in brackets after the procedure name. So our `DoConversion:` procedure will look something like this:

```
    PROC DoConversion:(Multiplier,FirstValue)
```

It's important to remember that these values are like temporary constants. You *cannot* make any changes to the information contained in these named values – they're for reference only.

We'll need somewhere to store the result:

```
            LOCAL StoreResult
```

Now we can do the maths itself:

```
            StoreResult=Multiplier*FirstValue
```

Finally we must return the result to the procedure that called the `DoConversion:` routine, and end this procedure:

```
        RETURN StoreResult
ENDP
```

Now, let's look back at the line that called this procedure:

```
ConversionReturn=DoConversion:(1.60934,ConversionValue)
```

The `RETURN`ed value of `StoreResult` is now going to be assigned to `ConversionReturn`. You'll notice that we call the same `DoConversion:` procedure for all the conversions; we just alter the values that we pass to it. This is an example of code re-use – we don't need to re-invent the wheel and duplicate the actual calculation code for every conversion.

Showing the Results

We've now done the conversion, and we want to display the result. We'll add this to the end of the `PROC DialogDistanceConversion:`. This will be through another dialog, using `dTEXT`. You should be able to follow this easily, but note we're using some text in the prompt field to help with the formatting:

```
dINIT "Conversion Complete"
IF ConversionDirection%=1
        dTEXT "Miles:",FIX$(ConversionValue,3,10),2
        dTEXT "Km:",FIX$(ConversionReturn,3,10),2
ELSE
        dTEXT "Km:",FIX$(ConversionValue,3,10),2
        dTEXT "Miles:",FIX$(ConversionReturn,3,10),2
ENDIF
dBUTTONS "OK",KdBUTTONEnter% OR KdBUTTONNoLabel% OR KdBUTTONPlainKey%
LOCK ON :DIALOG :LOCK OFF
```

Here's a good example of using an `IF` statement to help construct a dialog box. The program flow will only reach one pair of `dTEXT` commands. One of them shows miles to kilometers, the other shows the reverse.

`FIX$` is a built-in command to turn a number value into a string so it can be printed. The number is the first argument, then the number of decimal places (here limited to three), and finally the maximum length of the string (10 in this case). You should look this up, along with `NUM$` and `GEN$`, which perform similar functions.

`dBUTTONS` does exactly what it says, it displays a button, with text inside it, that can be pressed. When pressed it pretends to be a key,

which is then used to process the DIALOG command. The command list (as usual) has more details on any new command you will come across.

We're using the DIALOG command here without storing the returned value in a variable – because we don't need to know how (or why) the user dismisses this particular dialog box.

Extending Convert

As described in this chapter, Convert isn't quite complete. For example, the procedures for the other three types of conversion (short distances, weights and temperature) need to be written. There's also scope to not only add new conversions in the menu system, but to rework the dialog boxes to perhaps display a choice of more than two units (e.g. miles, kilometers, and furlongs).

What you should do now is complete Convert, and perhaps add in a feature or two of your own to ensure that you understand how Convert works, and how easy it is to (a) adapt it to your own needs and (b) create a quick and easy program that will process some information that is inputted by a user.

4.2 Summary

This chapter took what we learned previously about Event Core, and showed how to build a program around it. The last two main ways to interact with the user (dialogs and menus) were explained, and you saw how you can use the Appendix command list to look up and learn about new commands.

The 'Convert' program showed you how to use the menus and dialog to create your first useful program, and left you scope to extend it yourself.

5

Using Graphics in an Othello Game

Convert was your first practical OPL program, but it was limited to gathering information from the user and presenting new information back to them using the dialog box system. Our next program will use a graphical interface to gather and present information.

Gaming has advanced computing more than any other field. We're going to program the game Othello (the classic board game). During this you'll learn about handling graphics, reading in pen taps, controlling a cursor, and creating rules for a computer Othello player.

So, in the same way as we started Convert, let's create a new directory and a 'Source' folder, copy over the Event Core source code, and change the filename and constants to reflect that this will be "Othello".

We need to talk about four areas of OPL programming to put everything together:

- how OPL uses graphics
- representing an Othello board
- using pen taps or a cursor to read in the player's move
- programming rules to make the computer's move.

5.1 Using Graphics in OPL

Up until now, we've only used dialogs to show textual information. Almost every computer nowadays has the ability to display graphics (small pictures) on the screen. These can be used in games (e.g. to represent a ghost in Pac Man), or to make a clean and easy to use 'User Interface' with folders and files to open, just like you see on your Symbian OS phone.

We briefly touched on graphics in Chapter 3 when discussing the Event Core. As you may remember, the two main elements to graphical work in OPL are windows and bitmaps.

5.1.1 Windows

A window is where you will place your graphics. Consider it the sheet of paper you're going to work with. One of the great things about windows is that you can have more than one of them, so if your screen is split into two views, you would see a window on the left and a window on the right. You can also have hidden windows where you can draw, place graphics, etc. before showing it to the user all at once – like a finished picture.

5.1.2 Bitmaps

Your bitmap is the graphic that you create in a graphics or drawing package and include with your program. You can have lots of little pictures, or one big one and copy over only the part you need onto the paper (the window). The first thing you need to do is load the bitmap into the memory of the phone. Just because it has been copied onto the disk (e.g. when the user installed your program), that doesn't mean the program can use it directly. Instead, we call:

```
Id%(9)=gLOADBIT(Data$+MbmFiles$,0,3)
```

This will load the bitmap from the filename and path previously worked out in the Event Core code. Although we've briefly touched on loading bitmaps before, let's recap. The two numbers at the end are very important, and will change depending on the file. The second number (3) tells OPL what bitmap to use from the MBM file. MBM files can hold multiple bitmaps (hence .mbm). The first bitmap is bitmap 0, the second is bitmap 1, and so on.

The first number determines if you can alter or edit the bitmap within the program. Unless you are writing some kind of art package, or need to manipulate the .mbm for some reason, you would normally always leave this as **read only**, which requires the value 0; to be able to edit the bitmap, you would put a 1 here instead.

5.1.3 Closing Graphical Elements

Whenever you finish with a graphical window or bitmap, you should close it. This makes sure that the memory it occupied is reclaimed by the phone, and your program is more efficient. This is done simply with:

```
gCLOSE Id%(Foo%)
```

where `Foo%` is the array index number required. A good idea at the end of every program (in `PROC Exit:`) is to double check all the elements are closed:

```
Foo%=0
DO
        Foo%=Foo%+1
                IF Id%(Foo%) : TRAP gCLOSE Id%(Foo%)
                ENDIF
UNTIL Foo%=KMaxWindows%
```

TRAP will, of course, make sure an error is not raised if the graphical element isn't in use.

5.1.4 Copying MBMs to Windows

Let's break down the command that allows you to copy bitmaps to windows:

```
gUSE Id%(1)
gCOPY Id%(9),100,0,40,50,3
```

The first command (gUSE) tells OPL what graphical window is to be set as the current window. When you gCREATE a new window, it is automatically made the current (or active) window, but it is always best to use the gUSE command before any graphical operation to make sure you'll be working with the window you really want to.

Next is the gCOPY command. The first number, Id%(9), tells OPL which bitmap is to be copied into the current window. You now specify an area of that bitmap to copy. The next two numbers (100,0) say where the top left corner is (in pixels), measured from the top left corner of the bitmap overall bitmap. The next two numbers (40,50) specify the width and height of the portion of the bitmap which is to be copied, as in Figure 5.1. The final number determines how the bitmap is to be copied. Here's the different ways you can copy images. The checkerboard is

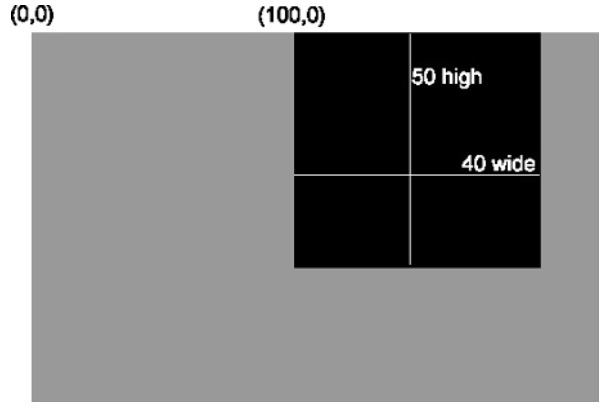


Figure 5.1 Width and height portion of bitmap to be copied

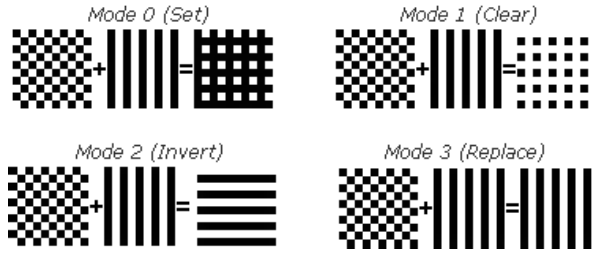


Figure 5.2 The four graphics modes

being copied in these examples onto the straight lines, and only the final number is changed to show you what you can do (see Figure 5.2).

5.1.5 Creating MBMs

As with our previous programs, let's do a sketch of what we want Othello to look like (see Figure 5.3).

From the above, we're now able to break it down into windows and bitmaps. The window part is the easiest. The outside status bars, titles, and CBA bars are something controlled by the runtime and so do not need dedicated graphical windows. Our main display can be represented by one window.

Looking at the bitmaps we'll need, we find the following are needed for Othello.

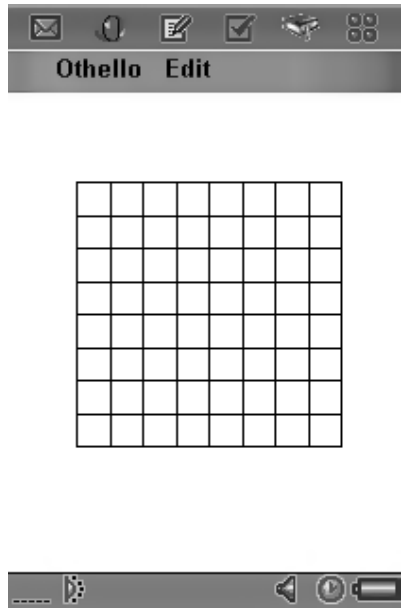


Figure 5.3 Othello empty

The Playing Pieces

Cunningly, we can store all our balls in one strip, along with an initial blank grid square in the first space, and a template (or 'mask') in the second space that we can use to effectively punch out a hole in the background so the individual balls can be copied over whilst preserving what's already there. This technique of using masks will be discussed later because it's really important and something you can use in pretty much any application you build.

The Cursor

Used to move around the grid so the user can see where the next piece will be placed.

The Othello Board

A completely empty board is saved in our MBM file and copied to the main window at the start of the game.

5.1.6 Creating the Bitmaps

Create your bitmaps in any graphics application (personally I use Microsoft Windows Paint because it's simple and gives you pixel perfect control – but anything similar will do). If you download the example files for this project from ***www.symbian.com/books/rmed/rmed-info.html*** the original bitmaps can be found in the `\Chapter5\Othello\Graphics\` folder, along with a text file and a small PC .exe file called BMConv.

5.1.7 Using BMConv

BMConv (Bitmap File Convertor) is a command line tool that will take a number of Microsoft Windows-format bitmap files, compile them into a single MBM file, and compress the resultant file for use on Symbian OS. While it is possible to input all the filenames of all your bitmaps on one command line, and not make any spelling mistakes, and get all the strings right first time, it is highly unlikely!

For this reason you can set up a small script in a text file that lists each command to be passed to BMConv on a separate line. This is the txt file in the `\Graphics\` folder for Othello.

A BMConv Script

Let's have a look at the BMConv script for Othello's three graphics:

```
Othello.mbm
/c12Pieces.bmp
/c12Cursor.bmp
/c12Board.bmp
```

The first line gives the filename of the MBM we want to make; in this case, Othello.mbm. We then list each bitmap file to be included on a separate line, and prefix the name of the bitmap file with the color depth that we want that graphic to be stored with in the MBM file.

In all these examples, we've used 12-bit color; /c for color, and 12 for 12-bit. You can equally use /c8 for 8-bit color, and /c16 for 16-bit color. You can also drop the c so /8 corresponds to 8-bit grayscale (which is useful for legacy devices, but not directly needed on current Symbian OS phones). Given that BMConv isn't great at scaling down the colors in your bitmaps, you should ensure you save them from your graphics package at the same depth as you specify in the BMConv script file.

Note that the bitmap files need to be in the same directory as BMConv.exe for this method to work.

Sending the MBM to the Phone

On the final distribution of any application you would package the MBM file into your SIS file, but while you are developing, this isn't always practical. Assuming you have planned your application correctly, you should only need to copy over the MBM file once.

You can get the file onto your phone using the same techniques as we discussed in Chapter 2 (Section 2.4.4) for transferring the .opo file.

5.2 Designing Othello

If you've never played Othello then here is a quick recap of the rules.

The object is to end the game with more of your pieces on the board than your opponent. Play starts with two pieces of each color in the center of an 8 × 8 board (see Figure 5.4). The two players take turns to capture their opponent's pieces by 'sandwiching' one or more opposing pieces in a straight line. When a piece is played at the end of a line of pieces that starts with another of his pieces, the pieces that have been 'sandwiched' are captured and change color to that player's color.

When there are no more moves possible for either player, or the board has no more empty squares, then the game is over and the player with the most of his own pieces on the board is the winner.

These rules are going to be incredibly useful in planning our computer version of Othello. This is a very structured game, and while it is easy to represent in a program, it still provides a very good challenge for the user.

At the start of any program, you need to sketch out how things are going to work and how things are done. That's a lot of things to cover. We've identified four areas, one of which (graphics) we've just looked at. Let's now look at the next three areas (representing the board, reading the player's move, making the computer's move), and then put it all together

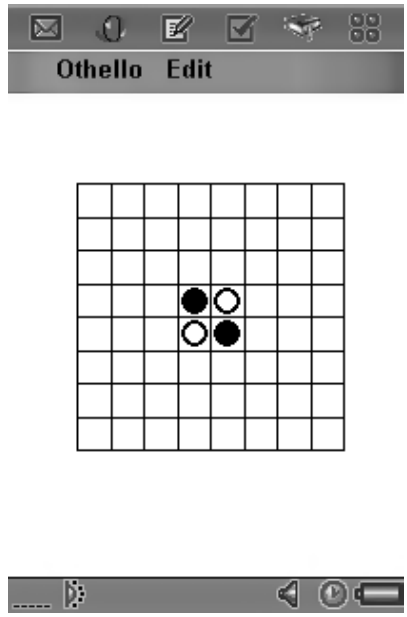


Figure 5.4 Othello board at start

to make our game. (As an aside, this is just a real-world example of breaking a large problem into smaller and smaller steps. Once you get to the smallest steps possible, it's those steps you use to write your OPL source code.)

5.3 Representing the Board

5.3.1 The Board as an Array

The Othello board is an 8 × 8 structure, so 64 squares in total. The easiest way to represent this is with an array called Board, with 64 elements. So right at the top of the skeleton code from Event Core, we'll add:

```
GLOBAL Board%(64)
```

Any newly declared variable is initially set to a zero value, so if we were to check any element of `Board%()` now, we would find it had a value of 0. Therefore, if a 'square' of the Othello board has no pieces on it, the value of the corresponding space in the array will be 0.

There are two other types of pieces on an Othello board, a white piece and a black piece. In our version, white will be the player and black will be the computer. These will be represented in the `Board%()` array by the number 1 (for the player) and 2 (for the computer).

5.3.2 Using Coordinates

OPL does not have two-dimensional arrays like other BASIC languages, so the board array is a strip of 64 'squares', while the Othello board is actually eight strips of eight. Now while we could just do a lot of hard maths all over the place to work out what square is what array box, we'll add two procedures to our code to simplify our array-based approach to the board:

```
PROC BoardIn:(Sx%,Sy%,Value%)
        rem *** Place Value% at (Sx,Sy) on Board (0,0 being top left)
        Board%((KBoardWidth%*Sy%)+(Sx%+1))=Value%
ENDP

PROC BoardOut%:(Sx%,Sy%)
LOCAL Foo%
        ONERR Marker::
        rem *** What is at (sx,sy) in the grid? RETURN this value.
        Foo%=Board%((KBoardWidth*Sy%)+(Sx%+1))
        ONERR OFF
        RETURN Foo%
        Marker::
        ONERR OFF
        RETURN 0
ENDP
```

When you call `BoardIn:`, you pass three numbers. The coordinates of a square on the board (and as noted in the `rem` comment, the top left square of the board is (0,0)) and the value you want to place in the grid position. For example, if you wanted to place a player piece (a value of 1) in the bottom right square, you would call the procedure with:

```
BoardIn:(7,7,1)
```

Discovering what piece is at a certain square is just as simple. You call `BoardOut%:` with the correct coordinate and the value of that square is `RETURN`ed. So after the previous example:

```
WhatIsHere%=BoardOut%:(7,7)
```

would `RETURN` the value of 1 in to `WhatIsHere%`.

Also note that rather than explicitly state the dimensions of the board, we use a constant to say it is eight squares wide. This is paired with another constant (can you guess) in the opening lines of the code:

```
CONST KBoardWidth%=8
CONST KBoardHeight%=8
```

The code for the board reading is now transportable, and can be dropped into any program you care to use. For example, a Go board can be 9, 13

or 17 squares wide. To represent this, all you do is change the constants. This is an example of how code can be re-used between more than one program, and a further benefit of using consistent programming style and variable names.

5.3.3 Setting up Variables and Arrays

The other main value we need to keep track of in the game is the score. In Othello, the score is simply the number of pieces that each player has on the board. These can be stored in two variables, which again should be globally declared:

```
GLOBAL PlayerScore%, ComputerScore%
```

There is also a constant called `KGridSize%`. This is the size (in pixels) of one square on the board. In more advanced programs, this could be a variable and one would calculate (when the app is initialized) the best size and create all the graphics at the time of starting the game. In Othello, we're not going down this road because it's needlessly complex for our example; we're going to use a fixed size on the grid size of 17 pixels, giving a grid of 136 pixels square, which comfortably fits on the screen of all Symbian OS phones:

```
CONST KGridSize%=17
CONST KGridWidth%=8
CONST KgridHeight%=8
```

Scaling the `KGridSize%` could be a good exercise once you have the basic Othello project working.

Finally, `KGridWidth%` and `KGridHeight%` say how many squares are in the grid on each side. Othello is played on an 8×8 grid.

5.4 Reading the Player's Move

As with any program, how you get information from the user is sometimes just as important as what you do with it. In Othello there are two main areas of interaction. The first will be a simple menu system that allows you to quit the current game and display the current scores. We've covered menus in the previous chapter, so check you can understand what's going on in the source code, as we won't go over it again here.

The second will be how to make a move. This is dependent on the target phone. The easiest way to make a move would be on a touchscreen, but devices using Series 60 will need to use a 'cursor and click' control.

5.4.1 Reading the (X,Y) from a Pen Tap

With a touch-sensitive screen, the easiest way to make a move is to tap the square that you want to place your piece on. Remember that we have a separate graphical window for just the board (where the top left (0,0) coordinate represents the very top left of the physical board). When a pen tap happens, the code flows from the main Event Loop procedure into the `PointerDriver:` procedure.

```
PROC PointerDriver:
        IF Ev&(4)=0 : rem Pen has been removed from the screen
               IF Ev&(3)=Id%(2) : rem Was it in the window with the
                   board?
                        FooX%=INT(Ev&(6)/GridSize%)
                        FooY%=INT(Ev&(7)/GridSize%)
                        FooValue%=BoardOut:(FooX%,FooY%)
                        MakePlayerMove:(FooX%,FooY%,1)
               ENDIF
        ENDIF
ENDP
```

We're using more of the values that are returned in the `Ev&` array (a full list of what is in each array element is in the appendix of this book. You should familiarize yourself with the `GETEVENT32` entry, which carries the full array explanation).

What we read into `FooX%` and `FooY%` is the actual pixel coordinates *in the window the pen was tapped*, **not** the actual coordinates of the entire screen. We then work out the coordinates of the grid square (rather than the pixels). Using the `INT` (integer) function means we will ignore the fraction of the answer, and take the whole number result only – which will be the square.

5.4.2 Using a Cursor with Keyboard Input

What is a Cursor?

You're probably familiar with the concept of a cursor. It's a highlighted area on a computer screen that shows where the next piece of information will be placed. In a word processing application, the vertical bar line is the cursor.

On the Othello board, we'll want to have a way of highlighting the square where the next piece will be placed. This is what the cursor will be for. To move the cursor, we'll use the joystick on the Series 60 devices (which has a push in function we'll use to let the player make the actual move). The Communicators also have a cursor pad and will use this same method, as the cursor pad on those devices sends the same keypress value as the Series 60 joysticks.

The cursor position will be stored as two coordinates, and these will be GLOBAL variables. As well as the current cursor position, we'll also need to know the previous cursor position. Why? Imagine the cursor drawn on the board. When we move to a new square, then we will draw the cursor on the new square. But what happens to the last drawn cursor on the previous square? It's still there. This is why we need to know the location of the previous cursor, so we can wipe it out. So we'll define:

```
GLOBAL CursorX%,CursorY%,OldX%,OldY%
```

Reading the Cursor Keys

The main work in reading the keys for moving the cursor is done inside the PROC KeyBoardDriver:. This is what we have from Event Core:

```
Mod%=Ev&(4) AND 255
IF Mod% AND 2 : Shift%=1  : ELSE : Shift%=0   : ENDIF
IF Mod% AND 4 : Control%=1 : ELSE : Control%=0 : ENDIF

rem Check for Direct keys here
IF Shift% : Key&=Key&-32 : ENDIF

rem Check for HotKeys here
IF Key&<=300
        ActionHotKey:
ENDIF
```

We're going to add the cursor keys into the "Direct Keys" section. But before we do that, there's one other thing to consider. What do we do when the cursor is at the edge of the board, and the player decides to try and move the cursor off the board?

Before changing the values of CursorX% and CursorY%, we'll use an IF statement to check if this will push the value to less than zero (off the top or left-hand edge) or if it will be larger than the width or height of the board (remember we listed the width and height of the board as constants – here's another place we use those values).

We'll also need to check for the 'fire' key being pressed to place a piece on the board. On Series 60 devices this is represented only by the CBA4 value, so we'll use the constant that holds the key value for CBA4 in the code. To make things a bit more like the built-in applications (always a good thing), we'll ensure that pressing the Enter key (commonly found on Communicators) will also place a piece:

```
rem Check for Direct keys here
IF Shift% : Key&=Key&-32 : ENDIF
IF Key&=KKeyCBA4& OR Key&=KKeySpace% OR Key&=KKeyEnter%
        MakePlayerMove:(CursorX%,CursorY%,1)
```

```
ELSEIF Key&=KKeyLeftArrow%
        CursorX%=CursorX%-1
        IF CursorX%<0
               CursorX%=0
        ELSE
               DrawCursor:
               OldX%=CursorX%
        ENDIF
ELSEIF Key&=KKeyRightArrow%
        CursorX%=CursorX%+1
        IF CursorX%>KBoardWidth%
               CursorX%=KBoardWidth%
        ELSE
               DrawCursor:
               OldX%=CursorX%
        ENDIF
ELSEIF Key&=KKeyUpArrow%
        CursorY%=CursorY%-1
        IF CursorY%<0
               CursorY%=0
        ELSE
               DrawCursor:
               OldY%=CursorY%
        ENDIF
ELSEIF Key&=KKeyDownArrow%
        CursorY%=CursorY%+1
        IF CursorY%>KBoardHeight%
               CursorY%=KBoardHeight%
        ELSE
               DrawCursor:
               OldY%=CursorY%
        ENDIF
```

These lines all follow the same principle of Event Core – when an event happens, you do something. Here it is either moving the cursor in one of four directions, or making a move. Once an event has happened, either we return to the loop and wait for another event, or the conditions for the end game have been met (either all the squares are full, or the player has resigned).

Is the Move Legal?

But how do we know if a move has been made successfully? Well, this comes down to thinking what you do when you make a move yourself. A legal Othello move is when you capture some of your opponent's pieces, by moving to a square not already occupied. This means that you need to sandwich at least one of your opponent's pieces between yours.

Let's work through this procedure:

```
PROC MakePlayerMove%:(X%,Y%,User%)
LOCAL DeltaX%,DeltaY%
```

The first step, though, is to check that the move is into an empty square (one where the `Board%()` value of the square equals zero). A simple call to the `BoardOut%:` procedure can do this:

```
IF BoardOut%:(X%,Y%)<>0
        RETURN 0
ENDIF
```

A simple test, and if the value returned is non-zero, then we'll leave this procedure through the `RETURN` system, and pass back the value of 0. This isn't used in the player checks, but will be used in the computer move routines (of which more later).

When we place a piece, it must have at least one opposition piece next to it. If you follow that line up, then it needs to come across another player's piece before reaching an empty square or the end of the board. The passed value `User%` lets us know whether we are testing for the player (`User%=1`) or the computer (`User%=2`):

```
LOCAL DeltaX%,DeltaY%,FooX%,FooY%,Opponent%,Captured%
...
IF User%=1
    Opponent%=2
ELSE
    Opponent%=1
ENDIF
DeltaX%=-2
DO
    DeltaX%=DeltaX%+1
    DeltaY%=-2
    DO
        DeltaY%=DeltaY%+1
        IF DeltaX%<>0 AND DeltaY%<>0
            rem Check Surrounding piece for opponent
            IF BoardOut%:(X%+DeltaX%,Y%+DeltaY%)=Opponent%
                DO
                    rem Found an opponent's piece, let's carry on up
                        the line
                    FooX%=X%+(2*DeltaX%) : FooY%=Y%+(2*DeltaY%)
                    IF BoardOut%:(FooX%,FooY%)=Opponent%
                        FooX%=FooX%+DeltaX% : FooY%=FooY%+DeltaY%
                    ELSEIF BoardOut%:(FooX%,FooY%)=User%
                        rem reverse direction and turn pieces over
                        DO
BoardIn:(FooX%,FooY%,User%)
                            Captured%=Captured%+1
                            FooX%=FooX%-DeltaX% :
                                FooY%=FooY%-DeltaY%
                        UNTIL (FooX%=X% AND FooY%=Y%)
                    ELSEIF BoardOut%:(FooX%,FooY%)=0
                        FooX%=-2 : REM Quick way to fulfill the UNTIL
                            conditions
                    ENDIF
```

```
                        UNTIL (FooX%=X% AND FooY%=Y%) OR FooX%=-1 OR
                            FooX%=KGridWidth% OR FooY%=-1 OR
                            FooY%=KGridHeight%
                ENDIF
            ENDIF
        UNTIL DeltaY%=1
    UNTIL DeltaX%=1
    RETURN Captured%
ENDP
```

We've two or three things going on in here. The principle is that we will check the eight possible directions that a line could result from. We use `DeltaX%` and `DeltaY%` for this – they show the change that needs to be applied to the coordinates of the nominated square (the delta) to check the direction. See Figure 5.5.

If we come across an opponent's piece, we'll follow that direction (using the delta values that represent the direction) down that line until we come across one of four things:

- another of our opponent's pieces – in which case we'll loop around again to check the next piece in line
- a blank square – in which case this line is not a sandwich, and doesn't count as part of a legal move. We stop checking down this line
- the edge of the board – which naturally stops us following this line
- we find a player's piece (a sandwich is made) – so this is a legal move, and we need to turn some pieces over. We begin a new loop, starting with this far end of the sandwich, working back to where the player made the move (the original `X%,Y%` coordinates), changing the value of all the pieces on the way in `Board%()` by using `BoardIn:(FooX%,FooY%,User%)`.

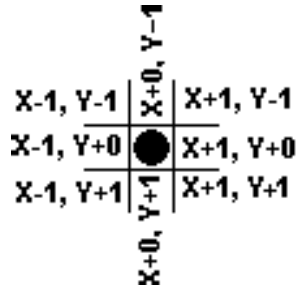


Figure 5.5 Delta values

At the end of this, we return the number of captured pieces. If it was an illegal move, then no pieces were captured and we `RETURN 0`. The routine calling `MakePlayerMove%:` will then know if it has to wait for another move to be made, or if the computer can make its move now.

Showing the Move

Now we've completed the move, and turned over any pieces that we've captured, we need to show this on the screen (remember, all we've been doing so far is changing values in the array). We'll set up two new procedures, one similar to `BoardIn:` that will display the piece in a selected square, and a second to step through all the squares in order:

```
PROC ShowBoard:
LOCAL FooX%,FooY%,FooPiece%
        PlayerScore%=0 : ComputerScore%=0
        FooX%=-1
        DO
               FooX%=FooX%+1
               FooY%=-1
               DO
                       FooY%=FooY%+1
                       FooPiece%=BoardOut%:(FooX%,FooY%)
                       ShowPiece:(FooX%,FooY%,FooPiece%)
                       IF FooPiece%=1
                              PlayerScore%=PlayerScore%+1
                       ELSEIF FooPiece%=2
                              ComputerScore%=ComputerScore%+1
                       ENDIF
               UNTIL FooY%=KGridHeight%-1
        UNTIL FooX%=KGridWidth%-1
        ShowAllScores:
ENDP

PROC ShowPiece:(X%,Y%,Type%)
        gUSE KBoardWindow%
        gAT X%*KGridSize%,Y%*KGridSize%
        gCOPY Id%(KPlayingPieces%,0,0,KGridSize%,KGridSize%,3)
ENDP
```

Yet again we're using the two nested `DO...UNTIL` loops to allow us to check through each square on the board. We use `KGridWidth%` and `KGridHeight%`, the constants we defined for the size of the board. Note that because we start at 0 and not 1, we need to subtract one to get 8 squares (square 0 to square 7). As we step through, we look up what piece is in that square (through the array), and then call the `ShowPiece%:`

routine. This will cause a ripple effect through the whole board, ending up with all the new pieces on display.

Cleaning up the Board

One advantage in a computer version of a board game such as Othello is that computers are great at keeping score. Inside the `ShowBoard:` procedure is an `IF...ELSEIF...ENDIF` statement, which counts the number of pieces the player and the computer have. These are `GLOBAL` variables, and when we call `PROC ShowAllScores:` at the end of this procedure, these will be displayed on the screen.

5.5 The Computer's Move – Doing A.I.

5.5.1 Creating Rules of Thumb

Making a computer play a good game is one of the most interesting challenges to a programmer. At each move, Othello offers a small number of choices. What we need to give the computer is a way to determine which choice is the best choice.

As with any programming task, you need to break down the task, and think about how you decide on your move. Because the aim of the game is to have as many pieces as possible at the end of the game, one valid strategy is to capture the most number of pieces on every move, and it is this 'rule' that we will use to make the computer's move.

5.5.2 A Simple Rule

So how do we do this? Well, how do you do it when you're playing? You look at every square you can make a move to and then count the number of pieces that you would capture. Then you make the move that captures the most. This is what we will program into the computer.

The Main Loop

```
PROC MakeComputerMove%:
LOCAL RecordMove%, WhichSquare%, BestMove%, BestMoveSquare%
        rem Go through all the squares...
        WhichSquare%=0
        DO
                WhichSquare%=WhichSquare%+1
                RecordMove%=CheckComputerMove%:(WhichSquare%)
                IF RecordMove%>BestMove%
                        BestMove%=RecordMove%
                        BestMoveSquare%=WhichSquare%
                ENDIF
```

```
            UNTIL WhichSquare%=KGridWidth%*KGridHeight%
             rem if there is no move, then pass.
            IF BestMove%<>0
                   PlayComputerMove:(BestMoveSquare%)
                   RETURN 1
            ELSE
                   dINIT "Pass"
                   dTEXT "","I have no move"
                   dBUTTONS "OK",KdBUTTONEnter%
                   LOCK ON
                   DIALOG
                   LOCK OFF
                   RETURN 0
            ENDIF
    ENDP
```

Checking Each Square's Value

So here we go through the 64 squares on the board, and each time through we call another procedure `CheckComputerMove%:`, which returns the number of pieces that would be captured if the move was made. This is similar to the `MakePlayerMove:` procedure:

```
PROC CheckComputerMove:(Target%)
    LOCAL DeltaX%,DeltaY%,FooX%,FooY%,X%,Y%,Captured%
    Y%=INT((Target%-1)/KGridWidth%)
    X%=(Target%-1)-(Y%*KGridWidth%)
    IF BoardOut:(X%,Y%)<>0
        RETURN 0
    ENDIF
    DeltaX%=-2
    DO
        DeltaX%=DeltaX%+1
        DeltaY%=-2
        DO
            DeltaY%=DeltaY%+1
            IF DeltaX%<>0 AND DeltaY%<>0
                rem Check Surrounding piece for opponent
                IF TBoardOut:(X%+DeltaX%,Y%+DeltaY%)=1
                    DO
                        rem Found an player's piece, let's carry on up the
                            line
                        FooX%=X%+(2*DeltaX%) : FooY%=Y%+(2*DeltaY%)
                        IF BoardOut:(FooX%,FooY%)=1
                             FooX%=FooX%+DeltaX% : FooY%=FooY%+DeltaY%
                        ELSEIF BoardOut:(FooX%,FooY%)=2
                             rem reverse direction - no need to turn
                                 pieces over
                             DO
                                  Captured%=Captured%+1
                                  FooX%=FooX%-DeltaX% :
                                      FooY%=FooY%-DeltaY%
                             UNTIL (FooX%=X% AND FooY%=Y%)
                        ELSEIF BoardOut:(FooX%,FooY%)=0
```

```
                            FooX%=-2 : rem Quick way to fulfill the UNTIL
                                conditions
                        ENDIF
                        UNTIL (FooX%=X% AND FooY%=Y%) OR FooX%=-1 OR
                            FooX%=KGridWidth% OR FooY%=-1 OR
                            FooY%=KGridHeight%
                ENDIF
            ENDIF
        UNTIL DeltaY%=1
    UNTIL DeltaX%=1
    RETURN Captured%
ENDP
```

There are only a few changes in this, compared to the one in the `MakePlayerMove:`. It's worth pointing out that while we still continue to use the main `Board%()` array, we don't actually change anything in the array (this is only done in our code through `BoardIn:`, which isn't called anywhere in the above code).

The first two lines calculate the (`X%,Y%`) coordinates of the square from the position in the array, and we then do a check to see if the square is occupied. If it is, there's no need to go through the long process of checking around the square as it's going to be impossible to play a piece there, hence the `RETURN 0` as soon as possible.

We then have an eerily similar procedure for `PlayComputerMove:`, but can dispense with the two `DO...UNTIL` loops as we know which square we'll be playing in (`BestMoveSquare%`).

And that's how the computer moves.

5.5.3 Mini-Max: Advanced A.I.

While we're not going to go into depth on actually programming the Mini-Max method of doing A.I., we'll talk about the principles here and leave the actual coding of the system as a project for you.

Mini-Max is named after the principle of minimizing your opponent's chances, while maximizing your own. We've actually already done half of this operation in our simple rule, where the computer always plays the best move available, maximizing its opportunity. What the computer fails to do here is to take into account what the board will look like for the player after their move. If you are playing Othello, you would visualize the board and make sure that your opponent doesn't have a 'killer' move available. You would make sure this isn't available – you're minimizing his opportunities.

So, ideally, once the computer has found a 'max' move, it needs to complete this move in a separate temporary array, and then check every resulting move available to the player after that. The best move

available to the player should be stored. We would do this for every computer move.

At the end of this, we'll have a list of moves and be able to see which one gives the best results for the computer. In Othello's case, we would see which computer move (coupled with the best player move) gives the computer the greatest advantage in the score.

Even looking two moves ahead (the computer and the player) you can see that the memory requirements to store all the information are much greater than for just looking at the best computer move. There is also a trade-off in speed, as we have many, many more states of the board to consider.

5.6 Putting it Together – the Main Game Loop

5.6.1 Initializing Everything

We've got a lot of bits of code now, all ready to be glued together. Before we do that, we need to set up a few things in our `PROC InitApp:`.

The main one here is to clear the board of pieces, and to reset the scores. While these will be at zero the first time the game is run, they need to be reset when a new game is started from within the application:

```
PROC InitApp:
        Foo%=0
        DO
               Foo%=Foo%+1
               Board%(Foo%)=0
        UNTIL Foo%=KGridWidth%*KGridHeight%
                PlayerScore%=0
        ComputerScore%=0
        CursorX%=0 : CursorY%=0
        OldX%=KGridWidth%+1, OldY%=KGridHeight%+1
ENDP
```

The initial cursor values are inside the visible window, but we don't call the procedure that displays the cursor until after a cursor key has been pressed. This is so that if the pen tap method is used by the player, the cursor is never displayed – but when any cursor key is pressed, up pops the cursor as expected. The old position is set outside the grid so it will not interfere with the drawing process.

5.6.2 Showing the Cursor

This is similar to `ShowPiece:`, but overlays the cursor on the square passed to this routine. Note the change in the graphics mode here, it

overwrites, not replaces, thus any playing piece under the cursor will still be seen:

```
PROC ShowCursor:(X%,Y%)
        gUSE KBoardWindow%
        gAT X%*KGridSize%,Y%*KGridSize%
        gCOPY Id%(KCursorPiece%,0,0,KGridSize%,KGridSize%,2)
ENDP
```

5.6.3 The Game Loop

Previously in Event Core, we simply looped around the call to read events in the main procedure. Here it is in plain English:

```
PROC Main:
        GLOBAL rem We'll fill these in as we plan the program
        LOCAL rem We'll fill these in as we plan the program
        Init:
        InitApp:
        DO
             rem Get a key press (an event)
             rem Act on this key press
        UNTIL rem we need to exit the program
        Exit:
ENDP
```

For Othello, we need to loop around the player move, and then the computer move, until all the squares are full:

```
PROC Main:
        GLOBAL rem We'll fill these in as we plan the program
        LOCAL rem We'll fill these in as we plan the program
        Init:
        InitApp:
        DO
            rem Player Move
            DO
                 rem Get a key press (an event)
                 rem Act on this key press
            UNTIL a move is made
            rem Computer Move
            rem Apply the A.I. rules
            rem make the move
       UNTIL all 64 squares are filled
       Show who the winner is
```

```
    Exit:
ENDP
```

This now needs to be translated into full OPL code, and all the elements we've discussed in this chapter need to be added in as well. You should now be able to do this with little assistance, although the full source code as always is on the supporting website ***www.symbian.com/books/rmed/rmed-info.html***

5.7 Summary

This chapter showed you how to use graphics in an OPL program, namely:

- creating the graphics on your PC
- using bmconv.exe to create Symbian mbm graphics files from windows bitmaps
- loading mbms into memory to use them in a program
- copying mbms in memory to a visible window.

We then started again with Event Core, and this time created a graphically rich game of Othello. We looked at reading pen taps on the screen, or reading in key presses to move a cursor. This way the program could be used on any platform.

We looked at representing a board and the pieces on the board in an array, how to read and act on this array, and how to display it. Using Mini-Max, you learned how to make a basic computer A.I. player and how to apply this to other games.

Finally, we showed you how to put this all together to make your Othello game.

6

Databases and a Notepad Program

For our final OPL project in this book, we will bring together elements from our previous three projects by building a Notepad program. Although a similar application may already be present in your phone, programming your own will allow you to use all your new OPL skills, from working through an interface, moving a cursor around information where there is more than one screen, accessing variables, presenting information, and making your own program look and feel like a built-in one.

It will also allow us to look in depth at using databases under OPL. Databases are powerful little creatures that can be used to store huge amounts of information that never change (like an encyclopedia), or rapidly changing information (like your diary/calendar). Learning to manipulate databases in OPL is an important step in creating genuinely useful programs.

6.1 What is a Database?

A database, like everything else on a computer, is a collection of 1s and 0s; don't forget that. The power in a database is how OPL can read and manipulate the information that is stored inside it.

One way of visualizing a database is to think of a pile of index cards representing the database, just as we did in Chapter 3 with the Event Core's INI file. Each index card has on it the same information in the margin. These are the headings. If you were creating a database of names and telephone numbers, then the two headings may be NAME and TEL. Note that the headings must be the same on every index card in the pile. On each card is written the name of one person, and one telephone number. In a database, this is called an entry, or a record.

How big is a database? As big as it needs to be is the short answer. In our analogy above, we can use as many index cards as we like, as long as the first one is always regarded as number one, and the rest are numbered in sequential order.

While we don't have the physical card holder in the file on our computer, we still have the database that holds all the records of information,

tied to their headings. Databases on a computer work in a very similar way to their real-life counterparts. (*Note*: You should know what we have here is a 'flat file' database, i.e. the structure is consistent. There are more complex databases called relational databases, but these are beyond the scope of this book.)

6.2 Our First OPL Database

6.2.1 The INI File

We've already created our first database, and many of the commands we're going to use have been seen already. Inside Event Core are all the relevant procedures to create, open, and save a simple database. So what are the differences between the INI database and the one we're going to create and use in this program? The main difference is that the Notepad database will have more than one record, so we'll need to be aware where we are in the database at any point in time; more on that in a moment. For now, let's have a look at the procedures and methods we're going to use on the Notepad database.

6.2.2 Construction

The headings in any program database need to be decided right at the start of the process – once we have `CREATE`d the database for the first time, we will not be able to change the headings. For our Notepad, this isn't too complicated, but you need to be aware of this issue when creating your own databases.

A Notepad database will have two fields. *Heading* (which will be displayed on the main screen for the user to choose) and *Text*, which is what will hold the actual information.

6.2.3 Creating or Opening our Notepad Database

In a similar way to the `LoadINI:` procedure, here's how we open our Notepad database:

```
PROC OpenFile:(Filename$)
        IF NOT EXIST(FileName$)
               CreateFile:(FileName$)
               RETURN
        ENDIF
        CloseFile:
        OPEN FileName$,B,Heading$,Text$
        FileOpen%=1
        rem Set Other Stuff
        TrueCursor%=1 : TopCursor%=1
ENDP
```

```
PROC CloseFile:
        IF FileOpen%=0 : RETURN : ENDIF
        TRAP USE B
        TRAP CLOSE
        FileOpen%=0
ENDP
```

This routine will commonly be called from inside the `InitApp:` procedure. In Notepad, we will only ever use one database, but it is possible to extend the program at a later date to be able to use more than one file, and switch between them as required.

Firstly in the code, we check to see if the file actually exists. If it does not (perhaps because this is the first time the program has been run), we jump to the `CreateFile:` procedure (which we'll come to in a moment).

Let's assume it does exist. Unlike the INI file, we have to keep our Notepad database open throughout the time the program is running, as it will need to be constantly read and edited. Therefore we have a variable (`FileOpen%`) to track this, where 0=closed and 1=open. We open the database, and assign it the reference letter "B" (remember that when we use the INI file, we use the letter "A"). After that we note what names we are going to use to look up the two fields, using as descriptive (to us) names as possible. We also set our two cursors to 1. We'll discuss why we have two cursors once we have all our database procedures in place.

`PROC CloseFile:` is simply a procedure to close the open database file referenced as B. Not strictly needed when we are hard-coding a single database to the program, but it will give you ideas for creating an app that can have more than one file. We have a check to ensure that the database is really closed (if it is, `FileOpen%` will equal zero, and we can safely return from this procedure). We then ensure we are using the correct database reference (`TRAP USE B`) and then `CLOSE` the file, `TRAP`ping any error if there is one. Finally we set the `FileOpen%` variable to zero:

```
PROC CreateFile:(FileName$)
                CloseFile:
                TRAP DELETE FileName$
                rem *** Create File Here
                CREATE FileName$,B,Heading$,Text$
                USE B
                rem Add Two 'default' records.

                INSERT
                B.Heading$="Welcome to Notepad"
                B.Text$="This is a simple database notepad application"
                PUT

                INSERT
                B.Heading$="Feel free to delete these"
```

```
                B.Text$="After all, this is your own app"
                PUT

                CLOSE
                OpenFile:(FileName$)
ENDP
```

In `CreateFile:`, we again check to see if the database already exists – something a multi-file program will need, but this means we can also use this to 'reset' our database from a menu option if we want to (i.e. start again from scratch). We then create the two initial entries into the Notepad so when the user opens the app they see something other than a white screen. Finally we `CLOSE` the database to ensure all the changes are saved, and then call `OpenFile:` to re-open the database.

6.2.4 The Position and Other Database Commands

One of the things you'll need to always keep in mind with your database is that each database can only reference one entry (with all the fields in that one entry) at a time. That entry will be whatever the current position in the database is. If your database is a stack of index cards, then the position is the index card that is currently on top of the pile.

You can find out where your position is in the database with

```
CurrentPosition%=POS
```

You can jump to a specified position:

```
POSITION NewPosition%
```

or move through the records one at a time, both forwards and backwards using

```
NEXT
```

and

```
BACK
```

Jumping the position to the start or end of the database can be done with

```
FIRST
```

and

```
LAST
```

All these commands (and other database commands) are, as usual, in the command list Appendix. They affect only the current database. To change to another database (if you have two or more databases open) use the `USE` command with the reference letter:

```
USE B
```

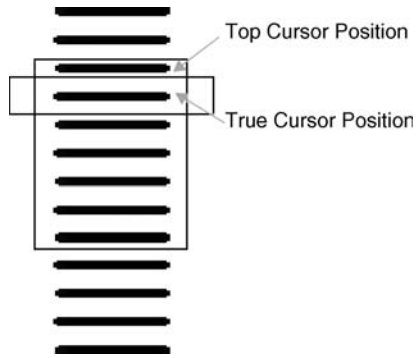
Whenever you first open a database, it is set to be the current database, which is why the INI procedures never really needed the `USE` command. However, here we will begin to introduce it anyway because we have more than one database – and of course it's good programming style!

6.2.5 Displaying a Cursor

In the Othello program, we had a constant representing the height of the playing squares so we knew just how many pixels we had to move the cursor. In Notepad, we will have a variable to represent this – because the fonts we use on each UI are different heights. We set this `PROC Init:` just after working out which UI we are running on.

Let's say (for example) that our screen can see seven lines of text. The cursor method we used in Othello will happily cope with seven lines of text, but what happens at eight lines of text? The cursor value would be increased to eight, and it would still be printed, but outside the visible area of the window.

We overcome this with a dual cursor method:



Three values are used, and are stored as `GLOBAL` variables:

```
GLOBAL TrueCursor%, TopCursor%, MaxCursor%
```

True Cursor

This is the cursor value that you are most familiar with. It represents which entry the cursor is on. So if you have 100 entries in the database,

and the cursor is on record 99, then the `TrueCursor%` will equal 99. `TrueCursor%` cannot have a value less than one or more than the total number of entries.

Top Cursor

In our diagram, the thin box over one line (which is a single entry in the database) represents the true cursor position. The box covering seven entries shows the seven entries that can be shown on the screen at any one time.

Going back to the question, what happens when we move the cursor down to the eighth entry? Well we simply shift this 'visible display box' down to make sure the eighth entry is inside the box. In other words, if `TrueCursor%` is going to drop out the visible display, then move down the visible display one entry.

To represent where this box is in the list of entries, our second cursor value is used. `TopCursor%` keeps track of the top entry in the visible display box.

Max Cursor

`TopCursor%` cannot have a value less than one, or allow itself to be moved so far down the entries that there are less than (in this example) seven entries in the display box. This job is helped by `MaxCursor%`, which is calculated in the `Init:` procedure. It holds the maximum number of entries that can be seen on the screen at any one time. In this case, `MaxCursor%=7`. How do we work it out?

```
MaxCursor%=INT(ScreenHeight%/FontHeight%)
```

This will be entered into `PROC InitApp:`, which we'll discuss once we've looked at all the individual elements.

Coding the Cursor Movement

The following lines need to be added into the `KeyboardDriver:` procedure:

```
ELSEIF Key&=KKeyUpArrow%
        rem CursorUp:
        IF TrueCursor%=1 : RETURN : ENDIF
        TrueCursor%=TrueCursor%-1
        IF TopCursor%>TrueCursor%
              TopCursor%=TrueCursor%
        ENDIF
        POSITION TrueCursor%
```

```
        ShowEntries:
ELSEIF Key&=KKeyDownArrow%
        IF TrueCursor%=COUNT : RETURN : ENDIF
        TrueCursor%=TrueCursor%+1
        IF EOF : TrueCursor%=TrueCursor%-1 : RETURN : ENDIF
        POSITION TrueCursor%
        IF TopCursor%+MaxCursor%<TrueCursor%
              TopCursor%=TopCursor%+1
        ENDIF
        ShowEntries:
```

As well as the similar commands to move the `TrueCursor%` that are familiar from Othello, we have added the code to move the 'window' of entries around depending on where the `TrueCursor%` lies.

On UIQ devices, you would also add in the `Key&` value for the scrolling wheel. These are `KKeyCBA1&` and `KKeyCBA2&`.

Displaying the Entries

Thanks to one of the cursor values, we know which database entry will be at the top of the screen (`TopCursor%`), and the number of entries we need to show (`MaxCursor%`). A simple `DO...UNTIL` loop will suffice:

```
PROC ShowEntries:
LOCAL FooY%,Gnu%
        FooY%=FontHeight%
        gUSE Id%(KMainWindow%)
        USE B : POSITION TopCursor%
        Gnu%=0
        DO
               Gnu%=Gnu%+1
               FooY%=FooY%+FontHeight%
               gAT 0,FooY%
               IF POS=TrueCursor% : gSTYLE 4 : ELSE : gSTYLE 0 : ENDIF
                      gPRINTB B.Heading$,Text$,2
               ENDIF
               NEXT
        UNTIL MaxCursor%=Gnu% OR EOF
        ShowEntries:
ENDP
```

`EOF`, end of file, will be true when the position in the database is at the very end of the database, i.e. if `NEXT` is called when there are no more entries. We call up the values of the heading and text fields by `B.Heading$` and `B.Text$`, which reference the database through the letter "B" and the fields by the names that were given to them *when the database was opened in this program*.

We only show the contents of the heading field in this main screen view.

Just before the `gPRINTB` command (first used back in the Event Core chapter), there is the command `gSTYLE`. A full breakdown of all the styles

can be found in the command list, but the IF statement here decides if the current entry is the entry with the cursor on it. If so, the entire entry is inversed to represent the cursor.

On moving the cursor (back in the KeyboardDriver: procedure) we simply ask the whole screen to be redrawn. It would be a simple exercise for the reader to optimize this code by only redrawing the whole screen if TopCursor% is altered – and only reprinting two entries for a normal cursor move.

6.2.6 Editing Entries

Editing an entry is a simple matter of pulling the two pieces of information out of the database using B.Heading$, placing these LOCAL string variables into Dialog, and then writing the changes back out to the database:

```
PROC EditNote:
LOCAL Foo%,Heading$(255),Text$(255),
    USE B
    POSITION TrueCursor%
    Heading$=B.Heading$
    Text$=B.Text$
    dINIT "Edit Note"
    dEDIT Heading$,"Heading",100
    dEDIT Text$,"Text",230
    dBUTTONS "Close",KdBUTTONEnter%
    LOCK ON
    Foo%=DIALOG
    LOCK OFF
    IF Foo%<>0
            MODIFY
            B.Heading$=Heading$
            B.Text$=Text$
            PUT
            ShowEntries:
    ENDIF
ENDP
```

The first half of this procedure should be recognizable from both the INI procedures from the Event Core, and constructing a Dialog from the Convert chapter.

Again, we're doing some defensive coding, and making sure that we assume nothing. That's why we make sure we are using the "B" database, and that the POSITION inside the database corresponds with the TrueCursor% position.

Once we exit the Dialog (and we've not left it with a cancel or an escape key press, which would see us RETURNed to the main program loop), we need to save the changes held in the LOCAL (and temporary) variables back into the database. Firstly, we let the database know that we are about to change something in the current record by using the

MODIFY command. We then list all the fields, including those where there are changes we want to make. Note that even if there is a field with no changes, we still need to include it here.

Once all our changes are in place, we then commit the changes to the database entry with the PUT command. Finally, in case the heading was changed, we re-display the screen by calling

```
ShowEntries:
```

Note that when you add or edit a record, this record goes to the end of the database, so your cursor may be in the same position, but it will be on a different record.

6.2.7 Adding Entries

Not surprisingly, adding a new record to a database is very similar to editing a record:

```
PROC AddNote:
LOCAL Foo%,Heading$(255),Text$(255)
    USE B
    dINIT "Create New Note"
    dEDIT Heading$,"Heading",100
    dEDIT Text$,"Text",230
    dBUTTONS "Close",KdBUTTONEnter%
    LOCK ON
    Foo%=DIALOG
    LOCK OFF
    IF Foo%<>0
            INSERT
            MODIFY
            B.Heading$=Heading$
            B.Text$=Text$
            PUT
            LAST
                            ShowEntries:
    ENDIF
ENDP
```

The first of the two changes is that we don't need to load in any existing values, because there aren't any. So once we make sure we're using the correct database, we can go straight into the Dialog box.

The second change is the use of the INSERT command, which, rather obviously, inserts a new blank record into the current database, which we can then MODIFY and PUT information into in the same way as when we were editing a record. Note that INSERT will add the blank record at the end of the database (like adding a new index card to the back of a pile), so your cursor will be in the same position at the end of this procedure, but on a different record.

6.2.8 Deleting Entries

The final operation we need to be able to do is to delete an entry from the database when we no longer want it:

```
PROC DeleteNote:
    USE B
    POSITION TrueCursor%
    rem *** Check if it can be deleted
    IF COUNT=1
            GIPRINT "Cannot Delete Last Note."
            RETURN
    ELSE
            ERASE
    ENDIF
    ShowEntries:
ENDP
```

As with our other 'editing' routines, we start by making sure we are at the correct position in the correct database. We then check to see if there is only one record in the database. While it is possible to write your program so it can handle the display and the cursor controls when there are no variables, that is left as an exercise for the reader. Here we have a simple one-line check that stops the user from deleting the last entry, using the `COUNT` command (which returns the number of entries in the database). If there is one record, then we `RETURN` with no changes (after displaying a status message using `GIPRINT`), otherwise we use the `ERASE` command. This acts on the entry at the current `POSITION` (which of course we made sure was matched up to the value of `TrueCursor%` at the start of the procedure).

When you delete a record, the position in the record is moved to the next entry in the database, which is also where `TrueCursor%` now points to. Note that if you make a lot of changes to the database (e.g. deleting many entries or editing a lot of data), the file size can sometimes grow as 'dead' space is left in the file on disk. This can be reclaimed immediately using the `COMPRESS` command. See the command listing for more details.

6.2.9 Putting Everything in Place

So now we've looked at all the code sections we'll need to write our Notepad program. Way back when we started the Event Core, we showed you that the key to making a program is breaking it down into sections – exactly what we've done here in looking at the things we need for the Notepad program.

What's left is to tie everything together in the main loop inside Event Core, and this means adding in the menu system and the hot keys (we've already shown the cursor controls you'll need).

The InitApp: Procedure

This is where you will make the main changes from the Event Core. Ask yourself what you need to do before you go into the main loop:

- open the database
- set the cursor position
- display the entries.

```
PROC InitApp:
       rem Load Database
       OpenFile:("C:\System\Apps\OPLPad\OPLPad.db")
       rem Set Cursor Positions and values
       TrueCursor%=1
       TopCursor%=1
       MaxCursor%=INT(ScreenHeight%/FontHeight%)
       rem Display Entries
       ShowEntries:
ENDP
```

Because `InitApp:` is called after `Init:`, the variables for the `ScreenHeight%` and `FontHeight%` will have been calculated when we checked the screen sizes (we'll add in the font information as we need it). What differentiates the `Init:` and `InitApp:` procedures? `InitApp:` may be called again during the running of the program, `Init:` cannot.

6.2.10 Adding Menus and Hot Keys

We'll be using the standard menu layout used in the Event Core, and adding in the options we'll need for the Notepad. We'll do this by adding in either a new menu card (for UIQ or Series 80 UI machines) or a cascading menu on a Series 60 machine.

We need to add in "New Note", "Edit Note" and "Delete Note". If you're adding this as a menu card, then something like:

```
mCARD "New Note",%n,"Edit Note",%a,"Delete Note",%d
```

and for Series 60 devices:

```
mCASC "Notes","New Note",%n,"Edit Note",%a,"Delete Note",%d
rem Call this Cascading menu from inside your menu system
```

You'd then add the three hot keys into the `ActionHotKey:` procedure, which call the procedures we've worked on above.

6.3 Summary

From the INI database in the Event Core (Chapter 3) we took these principles and introduced the OPL database commands:

- `POSITION`, `NEXT`, `BACK`, `FIRST`, `LAST` – used to move the database cursor around a database
- `USE` – switch between open databases
- `INSERT` – insert a new blank record at the end of a database
- `MODIFY` – change the contents of a record in a database
- `ERASE` – remove a record from a database.

Using these, and the idea of a cursor from Chapter 5, we looked at creating a Notepad style application that would be saved to disk so it is available when you open it again.

7

Publishing your OPL Application

OPL applications tend to be written for three main purposes. The first will be as a handy tool for yourself to use, in which case once you have the code working, you can quite happily carry on using it as is. The second will be if you are programming an application for use inside your own business environment – once it is coded, tested, and deployed to your users, you can move on to the next project. But a large number of OPL applications will be released on the Internet, some of them as freely available tools for other Symbian OS phone users, some as shareware applications, and some as commercial applications. In this chapter, we're going to look at the difference between these terms and how you can go about marketing your OPL applications to other Symbian OS users.

7.1 Types of Application

There are basically four ways you can make your OPL application available.

1. *Open Source:* You make the source code to your application available to other people who can then modify it or use it as an example for their own programs. There are various ways to do this and these come with too many intricacies to cover here – a quick Google search for 'open source' should give you lots of advice.
2. *Freeware:* These are full applications that do not require any payment to the author for usage. Freeware exists for a variety of reasons – for example, there is no longer any support from the author, the product has limited appeal, or the author is simply being philanthropic! As the author, you retain the copyright to the product and ownership of it, as well as the source code, but you make the binaries available, for free, for other people to use.
3. *Shareware:* This is the traditional 'try before you buy' concept, where you (as the author) provide a limited version of your application

to anyone who wants it. This is the working demonstration version which you should aim to make available to as many people as possible. What this version should do is make the user want to continue using it (or get access to more functionality). They do this by registering the software with you for a fee. This shareware registration fee is up to you, but generally basic games should start at around $5–$8, and a very detailed application (such as a complex Accounts Manager or Financial Planning tool) could go up to $25–$30. Getting the shareware fee right is a balancing act, and you should ask your testers what they think is a reasonable price once they're testing the final version. Similarly, deciding what works in the demonstration version is a balancing act too. Obviously, you don't want to put so many features in your demonstration version that people never register, but you want to make it attractive and indicative of the full application.

4. *Commercial:* Very similar to shareware, but normally produced by companies rather than individual developers. Commercial applications normally require the user to purchase them before use.

7.2 How Distribution Affects your Application Design

Testing

Once you've coded your application, you need to test it. This means trying every menu option and every hot key, in every combination. You should actually try to make the program crash. Not only that, but you should also try keys that you (as the programmer) know will not work.

It's a good idea to ask some other people to look over the program and test it as well, because they won't be as familiar with the application as you are, and may do things slightly differently. Before you release your application, you need to realize that your program will be running on other people's phones, and it must not have bugs (errors).

Bugs

The applications in this book aren't the most complicated, but even then you may have some bugs appearing when you go to expand the applications to try out your new skills. Bugs are a fact of life for programmers, and even with all your testing, it's probable that some bugs will be found after you release your application to the public. It's vital that once found, you correct them, and release an updated version of the application as soon as possible. It will give people confidence in you as a programmer, and ensure that anyone else who downloads and uses your application from then on doesn't come across the same old problems.

The First 20 Seconds

Something to bear in mind: while you may have the greatest application in the history of applications, it will be judged by the average user within 20 seconds of running. If it's not obvious what to do, if it looks horrible, then they won't continue looking at your program – and the user won't look at the program again. That's a lost registration or sale.

New Features

Very often, you will get emails from people saying that they think your application is wonderful, ''but wouldn't it be great if you could make it do this. . .'' For example, in our Othello application, you might want the option of having two human players, and using the program simply as an electronic board.

While you don't have to listen to these suggestions, if you get a lot of people asking for the same thing, and you're trying to promote your application as widely as possible (and gain shareware registrations or sales), then these feature requests are obviously for things in demand, and should help your sales if you add them.

Packaging your Application

It goes without saying that you should use a SIS file to put together all the files your application will need. But you shouldn't just present the end user with a SIS file. At the very least you need to have two files, the SIS and a Readme text file. The latter lets the user know how to install the SIS file (don't assume they know), what the application is, and your contact details. Both of these files should be placed in a standard ZIP file for distribution (which keeps the download size as small as possible).

Also remember that OPL applications require the OPL Runtime to be installed before they will work. At the very least, you should include a link in your Readme file to where users can download this. Alternatively, you can package the Runtime .SIS file as an 'embedded' file inside your own .SIS file – see Chapter 8 for more details.

Feedback

Whenever someone emails you, make sure to respond – even if it is simply to acknowledge that they've got in touch. Keeping in touch with people using your application will pay dividends in the long run. Keep track of all their emails, and ask (in the reply) if you'd like them to be emailed when a new version of the application (or a completely new application) is released.

In other words, like any business, make sure you keep your customers happy, make sure you can talk to them, and listen to their views.

7.3 How to Make your Application Available

7.3.1 Your Own Site

Finding web space on the Internet nowadays is not difficult. Most ISP hosting packages come with a few megabytes of web space you can use to upload the ZIP file, and host a few information pages about yourself, the application, and how to register or pay for it. At the very minimum you'll need to have the following pages:

- the main page, with links to all these other pages, and listing your latest new product and most recent updates to your applications
- one page for each of your applications
- a page where users can contact you, by a web form or your email address (be warned, putting an email address here means it will get spammed a lot, so you might not want to use your main email address).

7.3.2 Collecting Registration Fees

There are two main ways to collect the money from users. Both have advantages and disadvantages. You don't need to exclusively choose one or the other, you can use both. Browse them all and see which suits you.

PayPal (www.paypal.com)

This is an online service that allows small payments to be made by credit card between two people. It grew up from the Online Auction services, and it's perfect for collecting small shareware fees. There is a low commission rate, and you get the money almost immediately after your user sends it to you. Once you get the money, it is up to you to generate and send out either a registration code or the full version of your application to that user.

RegNet (www.reg.net)

Very similar to PayPal, RegNet allows you to create individual 'accounts' for your applications, each with its own unique link. Users can then visit the link you give them and pay by credit card for the product, resulting in an email to you so you can follow-up with them and provide a registration code or full version.

Handango and Others (www.handango.com/)

While there are quite a few online stores that allow you to post your shareware and freeware games, Handango is the largest and most recognizable. Once you sign up with them, you can upload to their server

both the demonstration version of your application, and either the full version or a registration code generation scheme. The advantage of this is that once you've uploaded everything to Handango, you don't need to do anything else in the sale process. They will process the money, move it to your account, and provide the user with what they need for the full application. Do note, however, that online stores can take a larger percentage of any shareware fees (up to 40%) and they may hold back sending you the fees until a certain minimum limit is reached (e.g. $100) or a certain period has elapsed (e.g. one quarter).

7.3.3 Programming in Limits for Demonstration Versions

There are lots of ways to build in limitations in your application, but one simple method that is easy to employ is to have a constant at the start of your source code:

```
CONST KRegistered%=0
```

When you compile your code, if `KRegistered%` is 0, then you are creating the demonstration version, and if it is 1, then it is the full version. You can then use this constant in your code to employ the main methods of limiting your demonstration version.

Using this method does mean that you will have two versions of the application, and you will need to provide anyone who registers the application with the full version. On sites such as Handango, you can upload a demonstration and a full copy, and they'll take care of it. If you're doing everything yourself, you'll need to email them (or make available) the full version as soon as possible after they register.

Nagging

You can make sure that someone using your demonstration version knows they are using the demonstration version. In small applications, this can sometimes be just enough to get people to register. For example, in `PROC Init:` you can add a simple `IF` statement around an information dialog:

```
...
IF KRegistered%=0
        dINIT "Please Register"
        dTEXT "","This version of "+KAppName$+" is",2
        dTEXT "","not registered. Please visit",2
        dTEXT "","http://www.yourwebsite.com/",2
        BUTTONS "OK",KdBUTTONEnter%
        DIALOG
ENDIF
...
```

Crippling

There may be one or two features in your application that aren't vital to the running of the application, but are useful – for example, the Undo feature in a game. This is a good way to put in a crippled feature. When someone tries to use a feature in the demonstration version that you want to cripple (i.e. make the user aware that if they buy the full version they can have this feature), an `IF` statement and the constant come into play again:

```
PROC UndoLastMove:
       IF KRegistered%=0
              dINIT "Disabled Feature"
              dTEXT "","This version of "+KAppName$+" is",2
              dTEXT "","not registered. Please visit",2
              dTEXT "","http://www.yourwebsite.com/",2
              BUTTONS "OK",KdBUTTONEnter%
              DIALOG
              RETURN
        ENDIF
        ...
ENDP
```

Always take care when using crippled functionality in an application. The end user must be able to judge the program fairly. There's no point in creating a great Notepad application, then crippling it by only letting the user save two notes. This means there's no way the user can judge the application before getting frustrated.

Maintaining a Single Version

Another approach to unlocking demonstration versions is to provide only a single program file, but make it respond to a registration code. The approach is very similar to the one outlined above, but instead of a `CONST`, you'd make `KRegistered%` a `GLOBAL` variable called `Registered%`, which is set to zero by default. You then offer the user a registration menu option which takes a code based on some set formula chosen by you and, if the correct code is entered, sets `Registered%` to 1 and persists this value in the INI file for future program launches. For example, a common approach is to generate a code based on the user's name to make it specific to them. Implementing this in code is too complex to go into here, but there are several examples available on the web, and many existing OPL authors are happy to offer you advice. See below for some links to websites with appropriate discussion forums.

When a user registers, you simply generate the appropriate code using the appropriate algorithm and send this to them – they can then unlock the version of the program already on their phone.

Warez and Pirates

A quick note about Warez – this is the practice of providing full versions of other people's applications for free. There's not much you can do to stop this, so the best advice I can give is to put some effort into protecting your application, but don't kill yourself over it – someone is going to pass the full version around at some point in time. As long as you make it very easy to find your demonstration version, and make it easy to purchase, you'll still gather registrations.

7.4 Promotion – Tell Everyone it's Available

7.4.1 Symbian Themed Websites

Once you release your application, you need to let people know that it is available. There are two main websites you should approach and get in touch with, both of which will help promote your site and application.

My-Symbian (http://mysymbian.com/)

Primarily a site for software, My-Symbian aims to host or link to every Symbian-based application out there. Contact details are shown on the front page, and generally your application will be posted in the front page news section, and in the main database after a few days of notifying them of its existence.

All About Symbian (www.allaboutsymbian.com/)

Mainly news articles and community information, All About Symbian do post news on applications, but do not post every application to the front page. They rely heavily on Handango to pass over details to their dedicated software section. Again, contact details are available on the front page.

Handango

If you are signed up with Handango, there are a number of Symbian sites on the Internet that will automatically pick up on the new and updated applications Handango host (for example, All About Symbian as mentioned above). This is a great way of getting your product out to more places.

Symbian Gear

Another online store, similar to Handango. Recently launched and therefore not as well known, as of the writing of this book. They have a huge amount of experience with their Palm OS and Microsoft Pocket PC stores.

7.4.2 Newsgroups and Forums

Most of the main websites (see above) have forums and bulletin boards. A short, polite note on these boards saying you have a new application will do wonders for your visibility to both users and other developers in the community.

7.5 Summary

Now you know about programming applications, this chapter took you through some common sense advice on distributing your applications. We looked at the things you need to consider when distributing your application, namely:

- test your program thoroughly
- make sure the first 20 seconds of use are perfect for a new user
- use a SIS file, and package the ZIP with everything required
- listen to feedback from your users and reply to everything.

We looked at registration fees and the best way to collect them if you decide to make your programs. You also need to think about how you will limit any demonstration application and how this works in your source code.

Finally, we talked about where you can go to host your program on the Internet, and the best places to promote your application.

8

Creating Applications and Installers

It's all well and good using file managers and Bluetooth to transfer files and run your OPL programs while developing your application, but when it comes to other people using them in the real world, there are more issues to consider. The two we will look at in this chapter are how they will get the application and all the files onto their phone in the correct place (installing it), and how they will run it when it is installed.

Symbian OS and OPL offer facilities to do both of these with very little work.

8.1 Creating an OPL Application

There are two considerations you need to have when making an OPL program (a .opo file) into a full application that is easy for users to install and run, without doing any of the manual file copying you as a developer have been doing so far. The first is to make sure your application behaves responsibly, and responds to all the requests from the phone. This has been taken care of already in our Event Core (where we talk about system commands and messages), so as long as you follow that method, then you do not need to worry about interacting with the system as a *bona fide* application.

The second is that you will need to uniquely identify your application in the Symbian OS environment, using a so-called 'UID'. This enables Symbian OS to keep track of your application individually, distinct from all the rest.

8.1.1 UID Numbers

The UID (unique identifier) number is a number that you obtain from the Symbian Developer Network. If you email ***uid@symbiandevnet.com*** and request a UID, you will be issued a UID at no cost, and this is your number forever, nobody else will be issued with it. Note that UID numbers less than 0x10000000 are never issued, and can be used as a

temporary stop-gap when testing or developing your application on your own phone.

When you do release your application, always double-check you use the official UID you have applied for, and not one of the low-numbered test UIDs.

8.1.2 Creating an Application from OPL Source Code

We create an OPL application (as opposed to a compiled .opo file) by adding in a new command at the very start of your code (before any `INCLUDE`s or `CONST` declarations). Here's what Event Core would have at the start:

```
APP Core,270488647
ENDA
```

The two elements following the `APP` command give the title of the application as it will appear in any Task Manager listings, and then the UID number. You can specify this in decimal (as above) or hex (using the standard OPL Hex format of &10000000). We then show we are finished defining the application with the `ENDA` command. Note the UID number here is Event Core's UID number, and you should change it before releasing your own applications.

When you compile OPL source code to an .opo file, the resulting .opo file is produced in the same directory as the source code. If you have added the `APP...ENDA` commands, then the compiled files will not appear in the same directory as the source code. They will be placed in the C:\ directory of your phone (or in the SDK Emulator File Structure if you are using OPLTran on the PC). The rule of thumb is that the compiled OPL application will always be placed in the correct place to be run. It will also create two files, rather than a single .opo file as before. Continuing to use Event Core as an example, on our phone (either in the emulator or on the device) you will find the following two files have been created:

```
C:\System\Apps\Core\Core.app
C:\System\Apps\Core\Core.aif
```

The actual compiled code is in the .app (`APP`.lication) file and is directly equivalent to any .opo code the same source code would generate without the `APP...ENDA` construct at the start.

The .aif is the application information file, and contains the UID, the caption for display on the system screen, and the details of the program icon that is shown on the system screen.

Figure 8.1 Graphic icon and mask

8.1.3 Creating an Icon for your Application

In the next chapter you'll see how to create a graphical file called an MBM (multiple bitmap file). You can create an application icon to be shown on the system screen by using the `ICON` command in the `APP...ENDA` structure:

```
APP Core,270488647
        ICON "CoreIcon.mbm"
ENDA
```

Each icon will need to have a graphic, and be accompanied by a mask, which is a computer graphics term for a silhouette. An icon and its mask are illustrated in Figure 8.1.

You'll also know that there are various zoom levels on the system screen. You will need to have an icon for each zoom level as appropriate for the phone you're targeting.

Under the UIQ interface, the icon sizes are 20 × 16 pixels and 32 × 32 pixels. The Standard UIQ color depth is 8 bits (256 colors), so use the /c8 (8-bit color) switch when creating your mbm file. Series 60 uses two sizes (44 × 44 for the large icon and 42 × 23) and Series 80 uses another two sizes (64 × 50 and 25 × 20 respectively). When you have more than one icon in your mbm file, you should order them {Icon, Mask, Icon, Mask} with the smallest size first, going to the largest size last.

8.2 Symbian Installation System – SIS Files

You can't ask the end user to move the individual files into the correct directory. We'd be looking (for a simple application) at three files (the .app, the .aif, and the .mbm file), and there would be a strong chance that the connectivity package they are using would not allow them the opportunity to see the underlying directory structure of their phone.

This is where we can harness the Symbian Installation System (SIS) method. On your PC you would gather together all the files that need to be installed, and run a command line tool, `makesis.exe`, that would put these together in one file (much like a ZIP file) along with a text file of instructions that say where all the files go on the phone, which the phone itself will read and follow like a script. This single file can then be sent to the phone through IrDA, Bluetooth, downloaded via WAP, emailed, or any other valid method. It should appear on the phone as a received file

in the messaging application. By selecting and running this file, the end user will install your application.

8.2.1 Putting Together your SIS Package File

Looking at our suggested directory structure on the PC, you'll see there is a directory called SIS. We're going to use this directory as we construct the SIS file.

Firstly, copy all the files that you need to be in the SIS file into this directory. We'll use the Event Core as our example application to create a SIS file. These files are:

```
Core.app
Core.aif
```

When we create the SIS file, we should use the full path name to these files on our PC, which gives us:

```
C:\OPL\Core\SIS\Core.app
C:\OPL\Core\SIS\Core.aif
```

We also need to know where to put these files on the phone when it is installed:

```
C:\System\Apps\Core\Core.app
C:\System\Apps\Core\Core.aif
```

This isn't exactly right, as applications should be able to be installed to any drive on the phone. When a SIS file is installed, the user is asked which drive to install the application onto (if more than one is available). This gives us the letter to use at the start of the path name, so we have a wild card symbol (an exclamation mark) for the path name:

```
!:\System\Apps\Core\Core.app
!:\System\Apps\Core\Core.aif
```

There may be circumstances when a file has to go onto the C:\ (e.g. you might provide an initial INI file that, by convention, must be on the C:\ drive). If this was the case, we'd have the following paths:

```
!:\System\Apps\Core\Core.app
!:\System\Apps\Core\Core.aif
C:\System\Apps\Core\Core.ini
```

The text file that goes into the SIS file is called the package file (.pkg) and contains the information we have gathered above:

```
"C:\System\Apps\Core\Core.app"-"!:\System\Apps\Core\Core.app"
"C:\System\Apps\Core\Core.aif"-"!:\System\Apps\Core\Core.aif"
```

The only thing we are missing now is the name of the application and the UID number.

Package Header Information

We now add the UID and the name of the application to the package file:

```
#{"Event Core"},(0x101F5447),1,0,0
"C:\System\Apps\Core\Core.app"-"!:\System\Apps\Core\Core.app"
"C:\System\Apps\Core\Core.aif"-"!:\System\Apps\Core\Core.aif"
```

The line we've added is made up of four sections, each doing a specific job in the installation process.

`#`: This signifies that the rest of this line will contain the application name, the UID number, and the three-part version number.

{"Event Core"}: This is the name of the application which will be shown to the user at install-time.

(0x101F5447): This is the UID number of the application. Note that this is the actual UID number of the Event Core application, so you should *not* use this number in your own applications. Also note the number is in hexadecimal format here.

1,0,0: This is the version number of your application. If you open up the Applications Manager on your phone, you'll see that each installed application also has a version number (you might need to look for a menu option for 'further details' on some phones). The three numbers show the major number, minor number, and build number. So 1,0,0 would be version 1.0(000). 2,51,188 would be versions 2.51(188). Note that if you want to make your application 1.10(188), you must specify 1,10,188 in your PKG file, rather than 1,1,188 – the latter corresponds to a version of 1.01(188) instead.

When you release an updated version of your application (perhaps because you found some bugs and have now fixed them) you should increment the major or minor numbers, and the build number, so that when the end users install the newer version, the phone knows it is a newer version.

Languages

Symbian OS is a multi-language operating system, so it is important to say which language your SIS file supports. While it is possible to have SIS

files support multiple languages, we're not looking at the more in-depth options for SIS files here, so we'll specify just one language, which is UK English.

We start a language option line with "&" and then the two-letter language code, in this case "EN" for UK English. A full list of language codes can be found in the documentation that came with the SDK you downloaded. While the docs will be primarily for C++ coders, there is a section for creating SIS files, and if you feel ready to make more complicated SIS files in the future, this is the best place to start.

Platform Dependency

One thing to remember with SIS files is that a SIS file can in theory be installed on any Symbian OS user interface platform, no matter which UI it uses (Series 60, UIQ, etc.). While OPL programs can be written that will run on all the platforms from one compiled version, there may be circumstances when you want to only have a SIS file installed if it is being targeted at a specific platform.

Each UI of Symbian has its own UID (after all, the UI is a type of program as well) and you can specify inside a SIS file to only install the file if you are on a certain UI version. For UIQ you would use the following command:

```
(0x101F617B),2,0,0,{"UIQ20ProductID"}
```

Here we see the UID number, the minimum version number that the target platform should be, and the application name. A full list of the UID numbers, versions, and names can be found in the SDK documentation.

If you do not have a dependency line, then two things will happen. Many phones will require a dependency line to be included, in which case your SIS file would not be installed, and an error message displayed to the user saying that an incorrect version is being installed will be shown.

Products that do not require a dependency will install the file, but there may be a conflict of interest here as a UIQ OPL application may not run correctly on a Nokia Communicator (depending on how you programmed it, of course).

Our Finished Package File

So here it is, our finished package file for Event Core. The final command shown here is the semi-colon, which acts just like rem in OPL. Anything on the line after the semi-colon is ignored, so it can be used to put in comments and *aide memoirs* as you go along:

```
; Package File For EVENT CORE.
;
; Specify the supported language.
&EN
;
; Installation name and header data
;
#{"Event Core"},(0x101F5447),1,0,0
;
<BEGINS>
;Files to remove on uninstallation
""-"C:\System\Apps\Core\Core.ini",FN
;
<ENDS>
; Files to install
; "Path on PC"-"Path on phone"
;
"C:\System\Apps\Core\Core.app"-"!:\System\Apps\Core\Core.app"
"C:\System\Apps\Core\Core.aif"-"!:\System\Apps\Core\Core.aif"
;
; Which UI Platform is this intended for
;
(0x101F617B),2,0,0,{"UIQ20ProductID"}
```

8.2.2 Generating the SIS File

Our SIS folder now has the following files:

```
Core.pkg
Core.app
Core.aif
```

The OPL SDK will have placed a copy of the command line utility `makesis.exe` in the correct directory path, so open a command line prompt and navigate to your SIS directory. Now use the command:

```
makesis Core.pkg Core.sis
```

This creates the SIS file Core.sis, which can be sent to the phone, packaged up in a ZIP file for downloading on the Internet, or whatever else is required of it.

8.2.3 Notes on SIS Files

The SIS file is a powerful facet of Symbian OS, and not every feature of SIS files has been covered here. The main area not covered here is security certificates. These provide a method to authenticate a SIS file using a pair of digital keys (one private, one public) and are used to identify the author of a SIS file, and to verify that a SIS file has not been tampered with since it was created.

Details on using certificates in SIS files can be found in the SDK documentation.

8.3 Summary

SIS files are the key to making a good user experience. They make your program much easier to install, and should be used whenever you distribute an application. We looked at how Symbian provides a unique number for every application, and how to include this in your SIS file.

We also looked at how to make an OPL program appear as an application on the main screen of icons on a phone, and how to include your own icon.

Finally, some extra features of SIS files (specifying languages and platform dependencies) were introduced.

9

Where Now With OPL?

Over the years, as we've seen, OPL has appeared on many different platforms, and over 2000 applications have been released online – with countless more being used as bespoke applications in business. Starting from a list of commands and basic principles, hundreds of authors have become programmers in their own right, using OPL on their pocket computers and smartphones. To give you an idea of what OPL can do, let's look at three applications written in OPL, to inspire you.

9.1 RMRBank, by Al Richey (RMR Software)

Al Richey has been around and seen almost every computer platform since the ZX81 and BBC B Micro in the early 1980s. He got to grips with OPL on the Psion Series 3a. Why? Because of the Money application that came with the Psion Series 3.

The first version of 'Psion Money' suited Al and his needs perfectly. He was also writing OPL programs to use himself to track shares, fuel consumption in his car, and the gas and electricity in his house. All these apps were originally just for his own use – exactly the type of thing OPL is so useful for. On moving up to the Series 3a, he also upgraded his copy of Money to the new computer. He was looking for a simple Home Accounts application, but this new version (along with the new 'Accounts and Expenses') insisted on double-entry bookkeeping, which he felt wasn't necessary. So he wrote his own, using OPL.

Al designed S3aBank (as it was originally called) to mirror the look and feel of the original Money program. After six months of working and tweaking it, he felt that S3aBank was good enough to be compared with the professional applications, and he released it onto the fledgling Internet, secretly hoping it would garner enough registrations to pay for the Series 3a. It is now recognized as one of the most popular pieces of OPL software ever, with over 10,000 sales across the myriad platforms it runs on. When the Psion Series 5 launched, S5Bank was available the same day. It was reworked with the Series 5mx, using the powerful new OPL Database Management System, and continued to sell very well.

RMRBank is now available for the Communicator and UIQ phones, and continues to be a popular application. After 10 years of coding, RMRBank is still doing exactly what its author needs, and is helping thousands of people manage their finances as well.

9.2 Fairway, by Steve Litchfield

Steve Litchfield is a full-time 'PDA Specialist' now, but back in the early 1990s he was still working for a living! Unable to be away from any type of computer over a two-week holiday in Scotland, Steve purchased a Psion Series 3. In those days, the paper manuals weighed more than the machine, and with the help of the enclosed OPL documentation, Steve had a working prototype of a golf game. Pitch and Putt, written purely for his own enjoyment, was uploaded to a few bulletin boards to see what sort of reaction it would get. It got a very good reaction, as Psion themselves got in touch with Steve, and 'Pitch and Putt' was packaged on the first Games Compilation disk released for the Series 3. Steve then explored the Shareware route, providing 'the first hole free' and asking for the registration fee to get the rest of the holes.

The success of this led to a full 18-hole tournament version. Called Fairway, it has been one of the most long-lived OPL games, going through three major versions, adding in features for digital sounds, to support different devices, and continuing to be marketed in official Games Compilations for the Series 3a and the Series 5 machines. Fairway now lives on in versions for both the Communicator and UIQ, in full color, ready to continue to provide an excellent game of golf in the mobile world.

9.3 EpocSync, by Malcolm Bryant

Malcolm Bryant isn't a programmer, but he's still created one of the most unusual OPL programs out there. Malcolm found that he was constantly beaming the same files between his Psion netBook and Revo during the day. So he decided to automate the process. EpocSync was the result. On lining up the infrared ports of the two machines and running EpocSync on both machines, they would connect to each other, and ensure that the latest copy of each file (in a list the user creates) is on each machine. EpocSync has the unusual distinction of being a program that has to talk to itself (when it is running on another machine). At each stage of the sync operation, each copy of the program needs to know where the other program is so the correct messages and packets of information can be sent between the machines.

After its release, EpocSync proved itself in two areas that Malcolm had never considered, but was then able to improve upon in subsequent releases. The first was that, because EpocSync was written completely in OPL, it would run happily on the Windows-based Symbian OS emulators. This meant that using EpocSync on both machines allowed users to synchronize the files on their Psion directly onto the PC, bypassing the connectivity suite. In essence, EpocSync and the SDK could be used as a 'Desktop PDA Companion' program. Finally, Malcolm added in support for FTP syncing with a third-party OPL extension. This used the same EpocSync engine, but now instead of copying the files to and from another Psion, the files were backed up to a regular FTP server on the Internet, providing remote backup for those on the move and travelling around a lot.

9.4 Final Summary... Moving Forwards Yourself

So now you have an understanding of OPL, what can you do? Well, as with any programming language, it's what you make of it that counts. You could build up a huge suite of applications for use inside your business, or in your personal life. You might just code something that already exists, but that's missing a key feature for you. And you may program and release apps on the Internet to gain recognition, or for profit. There are a large number of OPL programmers who have happily made a living programming in OPL – with even more Symbian OS phones out there, perhaps this could be you?

No matter what, OPL is now something you'll be able to use in your daily life. If you're unsure about something, then try it. If you need to ask for help, then the best starting point is the website for this book, or the OPL Forums on All About Symbian (***www.allaboutsymbian.com/***), My Symbian (***http://my-symbian.com/***), and the OPL Wiki Website (***www.allaboutopl.com/wiki***). And when you do release an application, be sure to send an email to myself and the OPL Development Team to let us know! You'll find details on how to do this at the above sites.

Part 2

Introduction to Part 2: Command Listing

As well as a structured introduction to programming, this book will continue to act as an ongoing reference as you program OPL in the months and years ahead. The core of OPL is the commands that are available in the runtime for you to use. No matter what version of OPL (UIQ, Series 60, or the Communicators) you use in the future, this command list will always remain current.

In fact, if you look back at the version of OPL that featured on the Psion Series 3 Organizer in 1992, almost all the commands available then are featured in the current Symbian OS runtimes.

Alphabetical Command Listing

Every OPL command is listed here, in a common format.

Command Name

This is the command name as typed into any OPL source code.

Command Syntax

A one-line example of the command, showing all the flags, variables, and their type, i.e. if they are integer variables, strings, etc. You may have the same command listed more than once. Some OPL commands have optional variables and flags, and each combination of these is listed. A good example of this is the banner print command, gPRINTB.

Short Description

A brief outline of the command and what it does.

Full Description

Some commands will need little more than the short description. Others need explaining in more depth. Here we break down each command, and show code examples where required.

Related Commands

As it says, OPL commands that are connected to the command you are looking up.

Const.oph Listing

The Const.oph file is a header file full of constants, that you can use to make your source code easier to read. More details on using constants can be found in Chapter 2. This list provides you with all the constants, and the values that they take, grouped by function.

Appendix 1

OPL Command List

(Derived from last Symbian OS v5 OPL Documentation, shipped with initial Series 80 OPL Release)

ABS — Absolute value of a floating point number

Usage: a=ABS(x)

Returns the absolute value of a floating point number, that is, without any +/− sign. For example ABS(−10.099) is 10.099.

If x is an integer, you won't get an error, but the result will be converted to floating point. For example ABS(−6) is 6.0. Use IABS to return the absolute value as a long integer.

ACOS — Arccosine

Usage: a=ACOS(x)

Returns the arc cosine, or inverse cosine (COS-1) of x.

x must be in the range −1 to +1. The number returned will be an angle in radians. To convert the angle to degrees, use the DEG function.

ADDR — Address of variable

Usage: a&=ADDR(variable)

Returns the address at which a variable is stored in memory.

The values of different types of variables are stored in bytes starting at ADDR(variable). See PEEK for details. The return type of this function should be a long integer, however, if the 64K memory limit is set via SETFLAGS, a& is guaranteed to fit into an integer.

See UADD, USUB.

ADJUSTALLOC Adjust memory allocation

Usage: pcelln&=ADJUSTALLOC(pcell&,off&,am&)

Opens or closes a gap at off& within the allocated cell pcell&, returning the new cell address or zero if out of memory. off& is 0 for the first byte in the cell. Opens a gap if the amount am& is positive, and closes it if negative. An error will be raised if the cell address argument is not in the range known by the heap.

See also SETFLAGS if you require a 64K limit to be enforced on 32-bit target devices. If the flag is set to restrict the limit, pcelln& is guaranteed to fit into an integer.

See ALLOC.

ALERT Alert dialog

Usage: any of

```
r%=ALERT(m1$,m2$,b1$,b2$,b3$)
r%=ALERT(m1$,m2$,b1$,b2$)
r%=ALERT(m1$,m2$,b1$)
r%=ALERT(m1$,m2$)
r%=ALERT(m1$)
```

Presents an alert – a simple dialog – with the messages and keys specified, and waits for a response. m1$ is the message to be displayed on the first line, and m2$ on the second line. If m2$ is not supplied or if it is a null string, the second message line is left blank.

Up to three keys may be used. b1$, b2$ and b3$ are the strings (usually words) to use over the keys. b1$ appears over an Esc key, b2$ over Enter, and b3$ over Space. This means you can have Esc, or Esc and Enter, or Esc, Enter and Space keys. If no key strings are supplied, the word CONTINUE is used above an Esc key.

The return value, r%, is one of the following:

KAlertEsc%	1	Escape key
KAlertEnter%	2	Enter key
KalertSpace%	3	Space bar

These constants are supplied in Const.oph.

ALLOC — Allocates a cell on the heap

Usage: pcell& = ALLOC(size&)

Allocates a cell on the heap of the specified size, returning the pointer to the cell or zero if there is not enough memory.

See also SETFLAGS if you require a 64K limit to be enforced on 32-bit target devices. If the flag is set to restrict the limit, pcelln& is guaranteed to fit into an integer.

See ADJUSTALLOC, REALLOC, FREEALLOC. See also Dynamic Memory Allocation.

APP — Defines an application

Usage:

```
APP caption,uid&
...
ENDA
```

Begins definition of an OPL application.

caption is the application's name (or caption) in the machine's default language. Note that although caption is a string, it is not enclosed in quotes.

uid& is the application's UID. For distributed applications, official reserved UIDs must be used. These can be obtained by contacting Symbian Ltd (see OPL applications for details of how to do this).

All information included in the APP...ENDA structure will be used to generate a. aif file, which specifies the applications caption in various languages, its icons for use on the System screen, and its setting of FLAGS.

See CAPTION, ICON, FLAGS.

APPEND — Adds a record to a data file

Usage: APPEND

Adds a new record to the end of the current data file. The record that was current is unaffected. The new record, the last in the file, becomes the current record.

The record added is made from the current values of the field variables A.field1$, A.field2$, and so on, of the current data file. If a field has not been assigned a value, zero will be assigned to it if it is a numeric field, or a null string if it is a string field.

Example:

```
PROC add:
  OPEN "address",A,f1$,f2$,f3$
  PRINT "ADD NEW RECORD"
  PRINT "Enter name:",
  INPUT A.f1$
  PRINT "Enter street:",
  INPUT A.f2$
  PRINT "Enter town:",
  INPUT A.f3$
  APPEND
  CLOSE
ENDP
```

To overwrite the current record with new field values, use UPDATE.

See Database File Handling for more details. See also INSERT, MODIFY, PUT, CANCEL, SETFLAGS.

ASC — Gets a character code from a string

Usage: a%=ASC(a$)

Returns the character code of the first character of a$. Alternatively, use A%=%char to find the code for char – e.g. %X for 'X'. If a$ is a null string ("") ASC returns the value 0.

Example: A%=ASC("hello") returns 104, the code for h.

ASIN — Arcsine

Usage: a=ASIN(x)

Returns the arc sine, or inverse sine (SIN-1) of x.

x must be in the range −1 to +1. The number returned will be an angle in radians. To convert the angle to degrees, use the DEG function.

AT — Positions the text cursor

Usage: AT x%,y%

Positions the cursor at x% characters across the text window and y% rows down. AT 1,1 always moves to the top left corner of the window. Initially, the window is the full size of the screen, but you can change its size and position with the SCREEN command.

A common use of AT is to display strings at particular positions in the text window. For example:

```
AT 5,2 :PRINT "message".
```

PRINT statements without an AT display at the left edge of the window on the line below the last PRINT statement (unless you use, or;) and strings displayed at the top of the window eventually scroll off as more strings are displayed at the bottom of the window.

Displayed strings always overwrite anything that is on the screen – they do not cause things below them on the screen to scroll down.

Example:

```
PROC records:
  LOCAL k%
  OPEN "clients",A,name$,tel$
  DO
    CLS
    AT 1,7
    PRINT "Press a key to"
    PRINT "step to next record"
    PRINT "or Q to quit"
    AT 2,3 :PRINT A.name$
    AT 2,4 :PRINT A.tel$
    NEXT
    IF EOF
      AT 1,6 :PRINT "EndOfFile"
      FIRST
    ENDIF
    k%=GET
  UNTIL k%=%Q OR k%=%q
  CLOSE
ENDP
```

ATAN — Arctangent

Usage: a=ATAN(x)

Returns the arc tangent, or inverse tangent (TAN-1) of x.

The number returned will be an angle in radians. To convert the angle to degrees, use the DEG function.

BACK — Moves back one record in the data file

Usage: BACK

Makes the previous record in the current data file the current record.

If the current record is the first record in the file, then the current record does not change.

BEEP Sounds the buzzer

Usage: BEEP time%,pitch%

Sounds the buzzer. The beep lasts for time%/32 seconds, so for a beep a second long make time%=32, etc. The maximum is 3840 (2 minutes).

The pitch (frequency) of the beep is 512/(pitch%+1) kHz.

BEEP 5,300 gives a comfortably pitched beep.

If you make time% negative, BEEP first checks whether the sound system is in use (perhaps by another OPL program) and returns if it is. Otherwise, BEEP waits until the sound system is free.

Example (a scale from middle C):

```
PROC scale:
 LOCAL freq,n%        REM n% relative to middle A
 n%=3          REM start at middle C
 WHILE n%<16
  freq=440*2**(n%/12.0)   REM middle A = freq
       440Hz
  BEEP 8,512000/freq-1.0
  n%=n%+1
  IF n%=4 OR n%=6 OR n%=9 OR n%=11 OR
       n%=13
    n%=n%+1
  ENDIF
 ENDWH
ENDP
```

Alternatively, sound the buzzer with this statement: PRINT CHR$(7). This produces a click sound.

Note: This command is deprecated from Symbian OS v6.0 onwards. Instead, developers should look at the MediaServer OPX or similar to generate sounds.

BEGINTRANS Begins a transaction on a database

Usage: BEGINTRANS

Begins a transaction on the current database. The purpose of this is to allow changes to a database to be committed in stages. Once a transaction has been started on a view (or table) then all database keywords will function as usual, but the changes to that view will not be made until COMMITTRANS is used.

See also COMMITTRANS, ROLLBACK, INTRANS.

BOOKMARK Places a bookmark on the current record

Usage: b%=BOOKMARK

Puts a bookmark at the current record of the current database view. The value returned can be used in GOTOMARK to make the record current again. Use KILLMARK to delete the bookmark.

BREAK Breaks out of a control loop

Usage: BREAK

Makes a program performing a DO...UNTIL or WHILE...ENDWH loop exit the loop and immediately execute the line following the UNTIL or ENDWH statement.

Example:

```
DO
 ...
  IF a=b
   BREAK
  ENDIF
 ...
UNTIL a=b
Rem BREAK command will take you to here
X%=3
```

BUSY Busy message

Usage: any of

```
BUSY str$,c%,delay%
BUSY str$,c%
BUSY str$
BUSY OFF
```

BUSY str$ displays str$ in the bottom left of the screen, until BUSY OFF is called. Use this to indicate 'Busy' messages, usually when an OPL program is going to be unresponsive to key presses for a while.

If c% is given, it controls the corner in which the message appears:

KBusyTopLeft%	0	top left
KBusyBottomLeft%	1	bottom left (default)
KBusyTopRight%	2	top right
KBusyBottomLeft%	3	bottom right

These corner value constants are supplied in Const.oph.

delay% specifies a delay time (in half seconds) before the message should be shown. Use this to prevent 'Busy' messages from continually appearing very briefly on the screen.

Only one message can be shown at a time. The maximum string length of a BUSY message is 80 characters, given by Const.oph's KBusyMaxtext%. An 'Invalid argument' error is returned for any value of str$ longer than this.

KBusyMaxtext%	80	maximum length of a BUSY message

BYREF — Passes a variable by reference

Usage: BYREF variable

Passes a variable by reference to an OPX procedure when used in a procedure argument list. This means that the value of the variable may be changed by the procedure.

See the OPX header files for more details.

CANCEL — Cancels a database transaction

Usage: CANCEL

Marks the end of a database's INSERT or MODIFY phase and discards the changes made during that phase.

CAPTION — Sets an application's caption

Usage: CAPTION caption$,languageCode%

Specifies an application's public name (or caption) for a particular language, which will appear below its icon on the Extras bar and in the list of 'Programs' in the 'New File' dialog (assuming the setting of FLAGS allows these) when the language is that used by the machine. CAPTION may only be used inside an APP...ENDA construct.

The language code specifies for which language variant the caption should be used, so that the caption need not be changed when used on a different language machine. If used, for whatever language, CAPTION causes the default caption given in the APP declaration to be discarded. Therefore CAPTION statements must be supplied for every language in which the application is liable to be used, including the language of the machine on which the application is originally developed.

The values of the language code should be one of the following:

KLangEnglish%	1	KLangPortuguese%	13
KLangFrench%	2	KLangTurkish%	14
KLangGerman%	3	KLangIcelandic%	15
KLangSpanish%	4	KLangRussian%	16
KLangItalian%	5	KLangHungarian%	17
KLangSwedish%	6	KLangDutch%	18
KLangDanish%	7	KLangBelgianFlemish%	19
KLangNorwegian%	8	KLangAustralian%	20
KLangFinnish%	9	KLangBelgianFrench%	21
KLangAmerican%	10	KLangAustrian%	22
KLangSwissFrench%	11	KLangNewZealand%	23
KLangSwissGerman%	12	KLangInternationalFrench%	24

These constants are supplied in Const.oph.

The maximum length of caption$ is 255 characters. However, you should bear in mind that a caption longer than around 8 characters will not fit neatly below the application's icon on the Extras bar.

See APP. See also OPL applications.

CHR$ Converts a character to a string

Usage: a$=CHR$(x%)

Returns the character with character code x%.

You can use it to display characters not easily available from the keyboard. For example, the instruction PRINT CHR$(133) displays an ellipsis (...).

CLEARFLAGS Clears system flags

Usage: CLEARFLAGS flags&

Clears the flags given in flags& if they have previously been set by SETFLAGS, returning to the default.

See SETFLAGS.

CLOSE Closes the current database view

Usage: CLOSE

Closes the current view on a database. If there are no other views open on the database then the database itself will be closed. See SETFLAGS for details of how to set auto-compaction on closing files.

If you've used ERASE to remove some records, CLOSE recovers the memory used by the deleted records, provided it is held either in the internal memory, or on a memory disk.

CLS Clears the text window

Usage: CLS

Clears the contents of the text window.

The cursor then goes to the beginning of the top line. If you have used CURSOR OFF the cursor is still positioned there, but is not displayed.

CMD$ Command line arguments

Usage: c$=CMD$(a%)

Returns the command line arguments passed when starting a program. Null strings may be returned. a% is one of the constant values KCmdAppName%, KCmdUsedFile%, or KCmdLetter% defined in Const.oph. CMD$(KCmdUsedFile%) and CMD$(KCmdLetter%) are only defined for OPL applications.

The value returned by CMD$ and its meaning depends on the value of a%:

KCmdAppName%	1	returns the full path name used to start running the program
KCmdUsedFile%	2	returns the full path name of the file to be used by an OPL application. For example, if CMD$(3)=R (see below), a default filename, including path, is passed in CMD$(2)
KCmdLetter%	3	returns one of the values below, and indicates the kind of command that was used to start the application

If an application passes KCmdLetter% to CMD$, the return value is one of:

KCmdLetterCreate$	"C"	application was started as a result of a Create new file command in the shell
KCmdLetterOpen$	"O"	application was started by opening a file belonging to it in the system screen
KCmdLetterRun$	"R"	application was run directly from the Extras bar, by opening the application itself in the system screen, or from the Program editor

The constants are defined in Const.oph.

See also GETCMD$ and OPL applications.

COMMITTRANS Commits the current database transaction

Usage: COMMITTRANS

Commits the transaction on the current view.

See also BEGINTRANS, ROLLBACK, INTRANS.

COMPACT Compacts a database file

Usage: COMPACT file$

Compacts the database file$, rewriting the file in place. All views on the database and hence the file itself should be closed before calling this command. This should not be done too often since it uses considerable processor power.

Compaction can also be done automatically on closing a file by setting the appropriate flag using SETFLAGS.

CONTINUE Jump to loop test condition

Usage: CONTINUE

Makes a program immediately go to the UNTIL... line of a DO...UNTIL loop or the WHILE... line of a WHILE...ENDWH loop, i.e. to the test condition. In this example the CONTINUE will take you back to the WHILE statement:

```
WHILE a<b
  ...
  IF a=c
   CONTINUE
  ENDIF
  ...
ENDWH
...
```

See also BREAK.

CONST Declares a constant

Usage: CONST KConstantName=constantValue

Declares constants that are treated as literals, not stored as data. The declarations must be made outside any procedure, usually at the beginning of the module. KConstantName has the normal type specification indicators (%, &, $, or nothing for floating point numbers). CONST values have global scope, and are not overridden by LOCAL or GLOBAL variables with the same name: in fact the translator will not allow the declaration new variables with a duplicate name. By convention, all constants should be named with a leading K to distinguish them from variables.

It should be noted that it is not possible to define constants with values smaller than −32768 (for integers) and −214748648 (for long integers) in decimals, but hexadecimal notation may be used instead (i.e. values of $8000 and &80000000, respectively).

COPY Copies files

Usage: COPY src$,dest$

Copies the file src$, which may be of any type, to the file dest$. Any existing file with the name dest$ is

deleted. You can copy across devices. You can also use wildcards if you wish to copy more than one file at a time.

If src$ contains wildcards, dest$ may specify either a filename similarly containing wildcards or just the device and directory to which the files are to be copied under their original names.

Example (to copy all the files from internal memory (in \opl) to d:\me\):

```
COPY "c:\opl\*","d:\me\"
```

(Remember the final backslash on the directory name.)

COS — Cosine

Usage: c=COS(x)

Returns the cosine of x, where x is an angle in radians.

To convert from degrees to radians, use the RAD function.

COUNT — Counts records in the current data file

Usage: c%=COUNT

Returns the number of records in the current data file. This number will be 0 if the file is empty.

If you try to count the number of records in a view while updating the view an 'Incompatible update mode' error will be raised (this will occur between assignment and APPEND/UPDATE or between MODIFY/INSERT and PUT).

CREATE — Creates a table in a database

Usage: CREATE tableSpec$,log,f1,f2,...

Creates a table in a database. The database is also created if necessary. Immediately after calling CREATE, the file and view (or table) is open and ready for access.

tableSpec$ contains the database filename and optionally a table name and the field names to be created within that table. For example:

```
CREATE "clients FIELDS name(40), tel TO phone", D,
    n$, t$
```

The filename is clients. The table to be created within the file is phone. The comma-separated list, between the keywords FIELDS and TO, specifies the field names whose types are specified by the field handles (i.e. n$, t$).

The name field has a length of 40 bytes, as specified within the brackets that follow it. The tel field has the default length of 255 bytes. This mechanism is necessary for creating some indexes. See dBase.opx – Database handling for more details on index creation.

The filename may be a full file specification of up to 255 characters. A field name may be up to a maximum of 64 characters long. Text fields have a default length of 255 bytes.

log specifies the logical filename A to Z. This is used as an abbreviation for the filename when you use other data file commands such as USE.

CURSOR Sets the text cursor

Usage: any of

```
CURSOR ON
CURSOR OFF
CURSOR id%
CURSOR id%,asc%,w%,h%
CURSOR id%,asc%,w%,h%,type%
```

CURSOR ON switches the text cursor on at the current cursor position. Initially, no cursor is displayed.

You can switch on a graphics cursor in a window by following CURSOR with the ID of the window. This replaces any text cursor. At the same time, you can also specify the cursor's shape, and its position relative to the baseline of text.

asc% is the ascent – the number of pixels (−128 to 127) by which the top of the cursor should be above the baseline of the current font. h% and w% (both from 0 to 255) are the cursor's height and width.

If you do not specify them, the following default values are used:

asc%	current font's ascent
h%	current font's height
w%	KCursorTypeNotFlashing%

If type% is given, it can have these effects:

KCursorTypeNotFlashing%	2	not flashing
KCursorTypeGrey%	4	grey

These constants are supplied in Const.oph.

You can add these values together to combine effects – if type% is 6 a grey non-flashing cursor is drawn. Using 1 for type% just displays a default graphics cursor, as though no type had been specified.

An error is raised if id% specifies a bitmap rather than a window.

CURSOR OFF switches off any cursor.

DATETOSECS Gets the number of seconds since 1/1/1970

Usage: s&=DATETOSECS(yr%,mo%,dy%,hr%,
mn%,sc%)

Returns the number of seconds since 00:00 on 1/1/1970 at the date/time specified.

Raises an error for dates before 1/1/1970.

The value returned is an unsigned long integer. (Values up to +2147483647, which is 03:14:07 on 19/1/2038, are returned as expected. Those from +2147483648 upwards are returned as negative numbers, starting from −2147483648 and increasing towards zero.)

See also SECSTODATE, HOUR, MINUTE, SECOND.

DATIM$ Current date and time

Usage: d$=DATIM$

Returns the current date and time from the system clock as a string – for example: "Fri 16 Oct 1992 16:25:30". The string returned always has this format – 3 mixed-case characters for the day, then a space, then 2 digits for the day of the month, and so on.

The string returned by DATIM$ can be parsed with MID$ using the following values for offsets within the string:

KDatimOffDayName%	1	offset of the day name (Fri)
KDatimOffDay%	5	offset of the day of the month (16)
KDatimOffMonth%	8	offset of the month name (Oct)
KDatimOffYear%	12	offset of the year number (1992)
KDatimOffHour%	17	offset of the hour (16)
KDatimOffMinute%	20	offset of the minute (25)
KDatimOffSecond%	23	offset of the second (30)

These constants are supplied in Const.oph.

Date.opx provides a large set of procedures for manipulating dates and for accurate timing.

DAY — Current day of the month

Usage: d%=DAY

Returns the current day of the month (1 to 31) from the system clock.

DAYNAME$ — Converts a number to a day name

Usage: d$=DAYNAME$(x%)

Converts x%, a number from 1 to 7, to the day of the week, expressed as a three-letter string. E.g. d$=DAYNAME$(1) returns MON.

Example:

```
PROC Birthday:
 LOCAL d&,m&,y&,dWk%
 DO
   dINIT
   dTEXT "","Date of birth",2
   dTEXT "","eg 23 12 1963",$202
   dLONG d&,"Day",1,31
   dLONG m&,"Month",1,12
   dLONG y&,"Year",1900,2155
   IF DIALOG=0 :BREAK :ENDIF
   dWk%=DOW(d&,m&,y&)
   CLS :PRINT DAYNAME$(dWk%),
   PRINT d&,m&,y&
   dINIT dTEXT "","Again?",$202
   dBUTTONS "No",%N,"Yes",%Y
 UNTIL DIALOG<>%y
ENDP
```

See also DOW.

DAYS Gets the number of days since 1/1/1900

Usage: d&=DAYS(day%,month%,year%)

Returns the number of days since 1/1/1900.
Use this to find out the number of days between two dates.
Example:

```
PROC deadline:
  LOCAL a%,b%,c%,deadlin&
  LOCAL today&,togo%
  PRINT "What day? (1-31)"
  INPUT a%
  PRINT "What month? (1-12)"
  INPUT b%
  PRINT "What year? (19??)"
  INPUT c%
  deadlin&=DAYS(a%,b%,1900+c%)
  today&=DAYS(DAY,MONTH,YEAR)
  togo%=deadlin&-today&
  PRINT togo%,"days to go"
  GET
ENDP
```

See also dDATE, SECSTODATE.
Date.opx provides a large set of procedures for manipulating dates and for accurate timing.

DAYSTODATE Converts the number of days since 1/1/1900 to a date

Usage: DAYSTODATE days&,year%,month%,day%

This converts days&, the number of days since 1/1/1900, to the corresponding date, returning the day of the month to day%, the month to month% and the year to year%. This is useful for converting the value set by dDATE, which also gives days since 1/1/1900.

dBUTTONS Defines exit keys

Usage: dBUTTONS p1$,k1%, p2$,k2%, p3$,k3%,...

Defines exit keys to go at the bottom or side of a dialog.
One or more exit keys can be defined with a p$,k% pair. Each pair specifies an exit key; p$ is the text to be displayed on it, while k% is the keycode of the shortcut

key. DIALOG returns the keycode of the key pressed (in lowercase for letters).

For alphabetic keys, use the % sign – %A means 'the code of A', and so on. The shortcut key is then Ctrl+alphabetic_key. Character codes lists the codes for keys (such as Tab) that are not part of the character set. If you use the code for one of these keys, its name (e.g. Tab, or Enter) will be shown in the key.

The following option flags can be applied to a button by adding the appropriate constant to the shortcut's keycode:

KDButtonNoLabel%	$100	button is displayed with no shortcut key label
KDButtonPlainKey%	$200	the key by itself – without the Ctrl modification – is used for the shortcut key

There are also constants for some key values:

KDButtonDel%	8	Del
KDButtonTab%	9	Tab
KDButtonEnter%	13	Enter
KDButtonEsc%	27	Esc
KDButtonSpace%	32	Space

These constants are supplied in Const.oph.

If a k% argument is negative, then the key is a 'Cancel' key. The corresponding positive value is used for the key to display and the value for DIALOG to return, but if you do press this key to exit, the var variables used in the commands like dEDIT, dTIME, etc. will not be set. You must negate the shortcut together with any added flags.

The Esc key will always cancel a dialog box, with DIALOG returning 0. If you want to show the Esc key as one of the exit keys, use −KDButtonEsc% as the k% argument so that the var variables will not be set if Esc is pressed.

There can be only one dBUTTONS item per dialog.

The buttons take up two lines on the screen. dBUTTONS may be used anywhere between dINIT and DIALOG; the position of its use does not affect the position of the buttons in the dialog.

This example presents a simple query, returning 'False' for No, or 'True' for Yes, providing shortcut keys

of N and Y, respectively and without labels beneath the keys:

```
PROC query:
  dINIT
  dTEXT "","FORGET CHANGES",2
  dTEXT "","Sure?",$202
  dBUTTONS "No",-(%N OR $300),"Yes",%Y OR $300
  RETURN DIALOG=%y
ENDP
```

See also dINIT.

dCHOICE Defines a choice list

Usage:

```
dCHOICE var choice%,p$,list$,matching%
```

or:

```
dCHOICE var choice%,p$,list1$+",..."
dCHOICE var choice%,"",list2$+",..."
...
dCHOICE var choice%,"",listN$
```

Defines a choice list to go in a dialog.

p$ will be displayed on the left side of the line. list$ should contain the possible choices, separated by commas – for example, "No,Yes". One of these will be displayed on the right side of the line, and the left and right arrows can be used to move between the choices.

choice% must be a LOCAL or a GLOBAL variable. It specifies which choice should initially be shown – 1 for the first choice, 2 for the second, and so on. When you finish using the dialog, choice% is given a value indicating which choice was selected – again, 1 for the first choice, and so on.

dCHOICE supports an unrestricted number of items (up to memory limits). To extend a dCHOICE list, add a comma after the last item on the line followed by "..." (three full stops), as shown in the usage above. choice% must be the same on all the lines, otherwise an error is raised. For example, the following specifies items i1, i2, i3, i4, i5, i6:

```
dCHOICE ch%,prompt$,"i1,i2,..."
dCHOICE ch%,"","i3,14,..."
dCHOICE ch%,"","i5,i6"
```

If matching% is set to true (KTrue% in Const.oph), incremental matching will be enabled on the choice list. If matching% is set to anything else OR omitted entirely from the dCHOICE line, the choice list will be constructed with incremental matching turned off.

See also dINIT.

dDATE — Defines a date edit box

Usage: dDATE var lg&,p$,min&,max&

Defines an edit box for a date, to go in a dialog.

p$ will be displayed on the left side of the line.

lg&, which must be a LOCAL or a GLOBAL variable, specifies the date to be shown initially. Although it will appear on the screen like a normal date, for example 15/03/92, lg& must be specified as "days since 1/1/1900".

min& and max& give the minimum and maximum values that are to be allowed. Again, these are in days since 1/1/1900. An error is raised if min& is higher than max&.

When you finish using the dialog, the date you entered is returned in lg&, in days since 1/1/1900.

The system setting determines whether years, months, or days are displayed first.

See also DAYS, SECSTODATE, DAYSTODATE, dINIT.

DECLARE EXTERNAL — Forces error reporting if procedures are used before they are declared

Usage: DECLARE EXTERNAL

Causes the translator to report an error if any variables or procedures are used before they are declared. It should be used at the beginning of the module to which it applies, before the first procedure. It is useful for detecting 'Undefined externals' errors at translate time rather than at runtime.

For example, with DECLARE EXTERNAL commented out, the following translates and raises the

error Undefined externals, i at runtime. Adding the declaration causes the error to be detected at translate time instead:

```
REM DECLARE EXTERNAL
PROC main:
  LOCAL i%
  i%=10
  PRINT i
  GET
ENDP
```

If you use this declaration, you will need to declare all subsequent variables and procedures used in the module, using EXTERNAL.

See also EXTERNAL.

DECLARE OPX Declares an OPX name

Usage:

```
DECLARE OPX opxname,opxUid&,opxVersion&
...
END DECLARE
```

Declares an OPX. opxname is the name of the OPX, opxUid& its UID, and opxVersion& its version number.

Declarations of the OPX's procedures should be made inside this structure.

dEDIT Defines a string edit box

Usage:

```
dEDIT var str$,p$,len%
```

or:

```
dEDIT var str$,p$
```

Defines a string edit box, to go in a dialog.

p$ will be displayed on the left side of the line.

str$ is the string variable to edit. Its initial contents will appear in the dialog. The length used when str$ was defined is the maximum length you can type in.

len%, if supplied, gives the width of the edit box (allowing for widest possible character in the font). The string will scroll inside the edit box, if necessary. If len% is not supplied, the edit box is made wide enough for the maximum width str$ could possibly be.

See also dTEXT.

dEDITMULTI Defines a multi-line edit box

Usage: dEDITMULTI var pData&,p$,widthInChars%, numLines%,maxLen%,readOnly%

Defines a multi-line edit box to go into a dialog. Normally the resulting text would be used in a subsequent dialog, saved to file, or printed using the Printer OPX (see Printer.opx – Printer and text handling). It is also possible to paste text into the buffer from other applications and vice versa, although any formatting or embedded objects contained in text pasted in will be removed.

pData& is the address of a buffer to take the edited data. It could be the address of an array as returned by ADDR, or of a heap cell as returned by ALLOC (see ADDR and ALLOC). The buffer may not be specified directly as a string and may not be read as such. Instead it should be peeked, byte by byte (see PEEK). The leading 4 bytes at ptrData& contain the initial number of bytes of data following. These bytes are also set by dEDITMULTI to the actual number of bytes edited. For this reason it is convenient to use a long integer array as the buffer, with at least 1+(maxLen%+3)/4 elements. The first element of the array then specifies the initial length.

If an allocated cell is used (probably because more than 64K is required), the first 4 bytes of the cell must be set to the initial length of the data. If this length is not set then an error will be raised. For example, if a cell of 100 000 bytes is allocated, you would need to poke a zero long integer in the start to specify that there is initially no text in the cell. For example:

```
p&=ALLOC(100000)
POKEL p&,0      REM Text starts at p&+4
```

Special characters such as line breaks and tab characters may appear in the buffer:

KParagraphDelimiter%	$06	paragraph delimiter
KLineBreak%	$07	line break
KPageBreak%	$08	page break
KTabCharacter%	$09	horizontal tab
KNonBreakingTab%	$0a	non-breaking horizontal tab
KNonBreakingHyphen%	$0b	non-breaking hyphen
KPotentialHyphen%	$0c	words will break here with a hyphen at the end of the line, if necessary
KNonBreakingSpace%	$10	non-breaking space
KPictureCharacter%	$0e	a picture
KVisibleSpaceCharacter%	$0f	visible space

These constants are supplied in Const.oph.

The prompt, p$ will be displayed on the left side of the edit box. widthInChars% specifies the width of the edit box within which the text is wrapped, using a notional average character width. The actual number of characters that will fit depends on the character widths, with e.g. more 'i's fitting than 'w's. numLines% specifies the number of full lines displayed. Any more lines will be scrolled. maxLen% specifies the length in bytes of the buffer provided (excluding the bytes used to store the length). readOnly% is an optional argument – if specified and set to true (KTrue% in Const.oph), the edit box will be made read only. If omitted or set to another value, the edit box will not be read only.

The Enter key is used by a multi-line edit box that has the focus before being offered to any buttons. This means that Enter can't be used to exit the dialog, unless another item is provided that can take the focus without using the Enter key. Normal practice is to provide a button that does not use the Enter key to exit a dialog whenever it contains a multi-line edit box. The Esc key will always cancel a dialog, however, even when it contains a multi-line edit box.

The following example presents a three-line edit box that is about 10 characters wide and allows up to 399 characters:

```
CONST KLenBuffer%=399
PROC dEditM:
  LOCAL buffer&(101)
  REM 101=1+(399+3)/4 in integer arithmetic
  LOCAL pLen&,pText&
```

```
LOCAL i%
LOCAL c%
pLen&=ADDR(buffer&(1))
pText&=ADDR(buffer&(2))
WHILE 1
  dINIT "Try dEditMulti"
  dEDITMULTI pLen&,"Prompt",10,3,KLenBuffer%
  dBUTTONS "Done",%d   REM button needed to
      exit dialog
  IF DIALOG=0 :BREAK :ENDIF
  PRINT "Length:";buffer&(1)
  PRINT "Text:"
  i%=0
  WHILE i%<buffer&(1)
   c%=PEEKB(pText&+i%)
   IF c%>=32
     PRINT CHR$(c%);
   ELSE
     REM just print a dot for special characters
     PRINT ".";
   ENDIF
   i%=i%+1
  ENDWH
 ENDWH
ENDP
```

See also dINIT.

DEFAULTWIN Changes the default window's color mode

Usage: DEFAULTWIN mode%

Changes the default window's color mode. The default window has ID=1, and uses a mode specific to the hardware capabilities of the Symbian OS phone it is running on. For example, a 4-grey window (2-bit, KColorDefWinWin4GrayMode%) is created on grey-scale machines, and a 256-color (8-bit, KColorDefWin256ColorMode%) window is created on color screen machines.

The default can be overridden using DEFAULTWIN. mode% specifies the new color mode:

KColorDefWin2GrayMode%	0	2-grey mode
KColorDefWin4GrayMode%	1	4-grey mode

KColorDefWin16GrayMode%	2	16-grey mode
KColorDefWin256GrayMode%	3	256-grey mode
KColorDefWin256ColorMode%	5	256-color mode

Note: The existing constants, KDefWin4ColorMode% and KDefWin16ColorMode%, are retained for backwards compatibility (see below).

These constants are supplied in Const.oph.

Using high-color mode uses more power than using modes with fewer colors.

You are advised to call DEFAULTWIN once near the start of your program and nowhere else if you need to change the color mode of the default window. If it fails with an 'Out of memory' error, the program can then exit cleanly without losing vital information.

DEG — Converts from radians to degrees

Usage: d=DEG(x)

Converts from radians to degrees.

Returns x, an angle in radians, as a number of degrees. The formula used is: 180*x/PI.

All the trigonometric functions (SIN, COS, etc.) work in radians, not degrees. You can use DEG to convert an angle returned by a trigonometric function back to degrees. For example:

```
PROC xarctan:
  LOCAL arg,angle
  PRINT "Enter argument:";
  INPUT arg
  PRINT "ARCTAN of",arg,"is"
  angle=ATAN(arg)
  PRINT angle,"radians"
  PRINT DEG(angle),"degrees"
  GET
ENDP
```

To convert from degrees to radians, use RAD.

DELETE — Deletes files

Usage: DELETE filename$

Deletes any type of file.

You can use wildcards, for example, to delete all the files in D:\OPL:

```
DELETE "D:\OPL\*"
```

See also RMDIR.

DELETE Deletes a table from a database

Usage: DELETE dbase$,table$

This deletes the table table$ from the database dbase$. To do this all views of the database, and hence the database itself, must be closed.

FILE Defines a filename edit box or selector

Usage:

```
dFILE var file$,p$,f%
```

or:

```
dFILE var file$,p$,f%, uid1&,uid2&,uid3&
```

Defines a filename edit box or selector, to go in a dialog. A 'Folder' and 'Disk' selector are automatically added on the following lines.

By default no prompts are displayed for the file, folder, and disk selectors. A comma-separated prompt list should be supplied. For example, for a filename editor with the standard prompts use:

```
dFILE f$,"File,Folder,Disk",1
```

Flags

f% controls the type of file editor or selector, and the kind of input allowed. You can add together any of the following values (from Const.oph):

–	0	use a selector
DFileEditBox%	1	use an edit box
KDFileAllowFolders%	2	allow directory names
KDFileFoldersOnly%	4	directory names only

DFileEditorDisallowExisting%	8	disallow existing files
KDFileQueryExisting%	16	query existing files
KDFileAllowNullStrings%	32	allows null string input
KDFileAllowWildCards%	128	obey/allow wildcards
KDFileSelectorWithRom%	256	allow ROM files to be selected
KDFileSelectorWithSystem%	512	allow files in the System folder to be selected

The first of the list is the most crucial. If you add 1 into f%, you will see a file edit box, as when creating a new file. If you do not add 1, you will see the 'matching file' selector, used when choosing an existing file.

If performing a 'copy to' operation, you might use 1+2+16 to specify a file edit box, in which you can type the name of a directory to copy to, and that will produce a query if you type the name of an existing file.

If asking for the name of a directory to remove, you might use 4, to allow an existing directory name only.

'Query existing' is ignored if 'disallow existing' is set. These two, as well as 'allow null string input', only work with file edit boxes, not 'matching file' selectors.

Restriction by UID

For file selectors, dFILE supports file restriction by UID, or by type from the user's point of view. Documents are identified by three UIDs, identifying which application created the document and what kind of file it is. Specifying all three UIDs will restrict the files as much as is possible, and specifying fewer will provide less restriction. You can supply 0 for uid1& and uid2& if you only want to restrict the list to uid3&. This may be useful when dealing with documents from one of your own applications: you can easily find out the third UID as it will be the UID you specified in the APP statement. Note that UIDs are ignored for editors. For example, if your application has UID KUidMyApp&, then the following will list only your application-specific documents:

```
dFILE f$,p$,f%,0,KUidOplDoc&,KUidMyApp&
REM KUidOplDoc& for OPL docs
```

Some OPL-related UID values are given in Const.oph.

KUidOplInterpreter&	268435575	the OPL interpreter
KUidOplApp&	268435572	an OPL app
KUidOplDoc&	268435573	an OPL document
KUidOPO&	268435571	an OPO
KUidOplFile&	268435594	an OPL file
KUidOpxDll&	268435549	an OPX

You can always press Tab to produce the full file selector with a dFILE item.

file$ must be declared to be KDFileNameLen% bytes long, since filenames may be up to this length. If it is shorter, an error will be raised.

KDFileNameLen% 255 maximum filename length for dFILE

See also dINIT.

dFLOAT Defines an edit box for a floating point number

Usage: dFLOAT var fp,p$,min,max

Defines an edit box for a floating point number, to go in a dialog.

p$ will be displayed on the left side of the line.

min and max give the minimum and maximum values that are to be allowed. An error is raised if min is higher than max.

fp must be a LOCAL or a GLOBAL variable. It specifies the value to be shown initially. When you finish using the dialog, the value you entered is returned in fp.

See also dINIT.

DIALOG Presents a dialog

Usage: n%=DIALOG

Presents the dialog prepared by dINIT and commands such as dTEXT and dCHOICE. If you complete the dialog by pressing Enter, your settings are stored in the variables specified in dLONG, dCHOICE, etc., although you can prevent this with dBUTTONS.

If you used dBUTTONS when preparing the dialog, the keycode that ended the dialog is returned. Otherwise, DIALOG returns the line number of the item that was current when Enter was pressed. The top item (or the title line, if present) has line number 1.

If you cancel the dialog by pressing Esc, the variables are not changed and KDlgCancel% is returned:

KDlgCancel%	0	return value: dialog was cancelled

See also dINIT.

dINIT Initializes a dialog

Usage: any of

```
dINIT
dINIT title$
dINIT title$,flags%
```

Prepares for definition of a dialog, canceling any existing one. Use dTEXT, dCHOICE, etc. to define each item in the dialog, then DIALOG to display the dialog.

If title$ is supplied, it will be displayed at the top of the dialog. Any supplied title$ will be positioned in a grey box at the top of the dialog. flags% can be any added combination of the following constants:

KDlgButRight%	buttons on the right rather than at the bottom
KDlgNoTitle%	no title bar
KDlgFillScreen%	use the full screen
KDlgNoDrag%	don't allow the dialog box to be dragged
KDlgDensePack%	pack the dialog contents (not buttons) densely

These constants are supplied in Const.oph. Sometimes these values will be ignored in certain UIs.

It should be noted that dialogs without titles cannot be dragged regardless of the 'No drag' setting. Dense packing enables more lines to fit on the screen for larger dialogs.

If an error occurs when adding an item to a dialog, the dialog is deleted and dINIT needs calling again. This is necessary to avoid having partially specified dialog lines. The following code will raise a 'Structure fault' error:

```
REM ** Faulty OPL fragment **
  dINIT
  ONERR e1
  REM bad arg list gives argument error:
```

```
  dCHOICE ch%, "ChList", "a,b,,,,c"
 e1::
  ONERR OFF
  dLONG l&,"Long",0,12345
  DIALOG
```

DIR$ Lists files that match a specification

Usage: d$=DIR$(filespec$)
then d$=DIR$("")

Lists filenames, including subdirectory names, matching a file specification. You can include wildcards in the file specification. If filespec$ is just a directory name, include the final backslash on the end, for example, "\TEMP\". Use the function like this:

DIR$(filespec$) returns the name of the first file matching the file specification
DIR$("") then returns the name of the second file in the directory
DIR$("") again returns the third, and so on

When there are no more matching files in the directory, DIR$("") returns a null string.

Example (listing all the files whose names begin with A in C:\ME\):

```
PROC dir:
  LOCAL d$(255)
  d$=DIR$("C:\ME\A*")
  WHILE d$<>""
    PRINT d$
    d$=DIR$("")
  ENDWH
  GET
ENDP
```

dLONG Defines an edit box for a long integer

Usage: dLONG var lg&,p$,min&,max&

Defines an edit box for a long integer, to go in a dialog.

p$ will be displayed on the left side of the line.

min& and max& give the minimum and maximum values that are to be allowed. An error is raised if min& is higher than max&.

lg& must be a LOCAL or a GLOBAL variable. It specifies the value to be shown initially. When you

finish using the dialog, the value you entered is returned in lg&.

See also dINIT.

DO...UNTIL Conditional loop

Usage:

```
DO
  statements...
  ...
UNTIL condition
```

DO forces the set of statements that follow it to execute repeatedly until the condition specified by UNTIL is met.

This is the easiest way to repeat an operation a certain number of times.

Every DO must have its matching UNTIL to end the loop.

If you set a condition that is never met, the program will go round and round, locked in the loop forever.

You can escape by pressing Ctrl+Esc, provided you haven't set ESCAPE OFF. If you have set ESCAPE OFF, you will have to return to the Task list, select your program in the list, and tap 'Close file'.

See also WHILE...ENDWH.

DOW Gets the day of the week from a date

Usage: d%=DOW(day%,month%,year%)

Returns the day of the week from 1 (Monday) to 7 (Sunday) given the date.

day% must be between 1 and 31, month% from 1 to 12, and year% from 1900 to 2155.

For example, D%=DOW(4,7,1992) returns KSaturday. Values for DOW are supplied in Const.oph:

KMonday%	1
KTuesday%	2
KWednesday%	3
KThursday%	4
KFriday%	5
KSaturday%	6
KSunday%	7

dPOSITION Positions a dialog

Usage: dPOSITION x%,y%

Positions a dialog. Use dPOSITION at any time between dINIT and DIALOG.

dPOSITION uses two integer values. The first specifies the horizontal position, and the second the vertical. dPOSITION −1,−1 positions to the top left of the screen; dPOSITION 1,1 to the bottom right; dPOSITION 0,0 to the center, the usual position for dialogs.

dPOSITION 1,0, for example, positions to the right-hand edge of the screen, and centers the dialog half-way up the screen.

Constants for these values are supplied in Const.oph.

See also dINIT.

dTEXT Defines text to be displayed in a dialog

Usage: dTEXT p$,body$,t%
or dTEXT p$,body$

Defines a line of text to be displayed in a dialog.

p$ will be displayed on the left side of the line, and body$ on the right side. If you only want to display a single string, use a null string ("") for p$, and pass the desired string in body$. It will then have the whole width of the dialog to itself. An error is raised if body$ is a null string and the text line is not a separator (see below).

body$ is normally displayed left-aligned (although usually in the right column). You can override this by specifying t%:

KDTextLeft%	0	left-align body$
KDTextRight%	1	right-align body$
KDTextCentre%	2	center body$

Alignment of body$ is only supported when p$ is null, with the body being left-aligned otherwise. In addition, you can add any or all of the following three values to t%, for these effects:

KDTextLineBelow%	$200	draw a line below this item
KDTextAllowSelection%	$400	allow this item's prompt (not its body text) to be selected

KDTextSeparator%	$800	specify this item as a text separator. p$ and body$ must both be the null string for this to take effect

The separator counts as an item in the value returned by DIALOG.

These constants are supplied in Const.oph.

See also dEDIT, dINIT.

dTIME Defines an edit box for a time

Usage: dTIME var lg&,p$,t%,min&,max&

Defines an edit box for a time, to go in a dialog.

p$ will be displayed on the left side of the line.

lg&, which must be a LOCAL or a GLOBAL variable, specifies the time to be shown initially. Although it will appear on the screen like a normal time, for example 18:27, lg& must be specified as seconds after 00:00. A value of 60 means one minute past midnight; 3600 means one o'clock, and so on.

min& and max& give the minimum and maximum values that are to be allowed. Again, these are in seconds after 00:00. An error is raised if min& is higher than max&.

When you finish using the dialog, the time you entered is returned in lg&, in seconds after 00:00.

The display in the time editor can be controlled via the t% argument. Add together one or more of the following constants from Const.oph to form t%:

KDTimeWithSeconds%	1	time editor shows seconds
KDTimeDuration%	2	editing a duration
KDTimeNoHours%	4	time editor does not show hours
KDTime24Hour%	8	time editor uses the 24-hour clock

This can be bulky to specify, however, so the following convenience constants are also defined:

KDTimeAbsNoSecs%	0	absolute + no seconds
KDTimeAbsWithSecs%	1	absolute + seconds
KDTimeDurationNoSecs%	2	duration + no seconds
KDTimeDurationWithSecs%	3	duration + seconds

For example, 03:45 represents an absolute time, while 3 hours 45 minutes represents a duration.

Absolute times are displayed in 24-hour or a.m./p.m. format according to the current system setting. 8 displays the time in 24-hour clock, regardless of the system setting.

Absolute times always display a.m. or p.m. as appropriate, unless the 24-hour clock is being used. Durations never display a.m. or p.m. Note, however, that if you use the flag 4 (no hours) then the a.m./p.m. symbol will be displayed and the flag 2 must be added if you wish to hide it.

See also dINIT.

dXINPUT Defines an exit box for a secret string

Usage: dXINPUT var str$,p$,seed%

Defines a secret string edit box, such as for a password, to go in a dialog.

p$ will be displayed on the left side of the line.

str$ is the string variable to take the string you type.

KDXInputMaxLen%	16	maximum length of str$

This constant is supplied in Const.oph.

Initially the dialog does not show any characters for the string unless seed% is set to true (KTrue% in Const.oph); if seed% is omitted or set to another value, the initial contents of str$ are ignored. A special symbol will be displayed for each character you type, to preserve the secrecy of the string.

See also dINIT.

EDIT Displays a string for editing

Usage: EDIT a$

Displays a string variable that you can edit directly on the screen. All the usual editing keys are available: the arrow keys move along the line, Esc clears the line, and so on.

When you have finished editing, press Enter to confirm the changes. If you press Enter before you have made any changes, then the string will be unaltered.

If you use EDIT in conjunction with a PRINT statement, use a comma at the end of the PRINT statement,

so that the string to be edited appears on the same line as the displayed string:

```
...
PRINT "Edit address: ",
EDIT A.address$
UPDATE
....
```

TRAP EDIT

If the Esc key is pressed while no text is on the input line, the 'Escape key pressed' error (−144) will be returned by ERR provided that the EDIT has been trapped. You can use this feature to enable the user to press the Esc key to escape from inputting a string.

See also INPUT, dEDIT.

ELSE/ELSEIF/ENDIF See IF

See IF.

ENDA See APP

See APP.

ENDV See VECTOR

See VECTOR.

ENDWH See WHILE

See WHILE.

EOF Checks for end-of-file

Usage: e%=EOF

Finds out whether you're at the end of a file yet.

Returns −1 (true) if the end of the file has been reached, or 0 (false) if it hasn't.

When reading records from a file, you should test whether there are still records left to read, otherwise you may get an error.

Example:

```
PROC eoftest:
  OPEN "myfile",A,a$,b%
  DO
```

```
    PRINT A.a$
    PRINT A.b%
    NEXT
    PAUSE -40
  UNTIL EOF
  PRINT "The last record"
  GET
  RETURN
ENDP
```

ERASE — Erases a record in the current data file

Usage: ERASE

Erases the current record in the current file.

The next record is then current. If the erased record was the last record in a file, then following this command the current record will be null and EOF will return true.

ERR — Number of last error

Usage: e%=ERR

Returns the number of the last error which occurred, or 0 if there has been no error.

Example:

```
  ...
  PRINT "Enter age in years"
age::
  TRAP INPUT age%
  IF ERR=-1
    PRINT "Number please:"
    GOTO age
  ENDIF
  ...
```

You can set the value returned by ERR to 0 (or any other value) by using TRAP RAISE 0. This is useful for clearing ERR.

See also ERR$, ERRX$. See Runtime errors – Handling errors reported while running programs for full details, and OPL error values for the list of error numbers and messages.

ERR$ — Looks up an error message by number

Usage: e$=ERR$(x%)

Returns the error message for the specified error code x%.

ERR$(ERR) gives the message for the last error that occurred. Example:

```
TRAP OPEN "\FILE",A,field1$
IF ERR
  PRINT ERR$(ERR)
  RETURN
ENDIF
```

See also ERR, ERRX$. See Runtime errors – Handling errors reported while running programs for full details, and OPL error values for the list of error numbers and messages.

ERRX$ Gets an extended error message

Usage: x$=ERRX$

Returns the current extended error message (when an error has been trapped), e.g.

```
'Error in
    MODULE\PROCEDURE,EXTERN1,EXTERN2,...'
```

which would have been presented as an alert if the error had not been trapped. This allows the list of missing externals, missing procedure names, etc. to be found when an error has been trapped by a handler.

See Runtime errors – Handling errors reported while running programs for full details, and OPL error values for the list of error numbers and messages.

ESCAPE OFF Disables Ctrl+Esc

Usage:

```
ESCAPE OFF
...
ESCAPE ON
```

ESCAPE OFF stops Ctrl+Esc being used to break out of the program when it is running. ESCAPE ON enables this feature again.

ESCAPE OFF takes effect only in the procedure in which it occurs, and in any subprocedures that are

called. Ctrl+Esc is always enabled when a program begins running.

If your program enters a loop that has no logical exit, and ESCAPE OFF has been used, you will have to go to the Task list, move to the program name, and select Close file.

EVAL — Evaluates a mathematical expression

Usage: d=EVAL(s$)

Evaluates the mathematical string expression s$ and returns the floating point result. s$ may include any mathematical function or operator. Note that floating point arithmetic is always performed.

EVAL runs in the "context" of the current procedure, so globals and externals can be used in s$, procedures in loaded modules can be called, and the current values of gX and gY can be used, etc. LOCAL variables cannot be used in s$ (because the translator cannot deference them).

For example:

```
DO
  AT 10,5 :PRINT "Calc:",
  TRAP INPUT n$
  IF n$="" :CONTINUE :ENDIF
  IF ERR=-114 :BREAK :ENDIF
  CLS :AT 10,4
  PRINT n$;"=";EVAL(n$)
UNTIL 0
```

See also VAL.

EXIST — Checks if a file exists

Usage: e%=EXIST(filename$)

Checks to see that a file exists. Returns KTrue% if the file exists and KFalse% if it doesn't.

Use this function when creating a file to check that a file of the same name does not already exist, or when opening a file to check that it has already been created:

```
IF NOT EXIST("CLIENTS")
  CREATE "CLIENTS",A,names$
ELSE
```

```
 OPEN "CLIENTS",A,names$
ENDIF
...
```

EXP — Exponential

Usage: e=EXP(x)

Returns ex – that is, the value of the arithmetic constant e (2.71828...) raised to the power of x.

EXTERNAL — Declares procedure prototypes and external variables

Usage: EXTERNAL variable
or EXTERNAL prototype

Required if DECLARE EXTERNAL is specified in the module.

The first usage declares a variable as external. For example, EXTERNAL screenHeight%.

The second usage declares the prototype of a procedure (prototype includes the final : and the argument list). The procedure may then be referred to before it is defined. This allows parameter type-checking to be performed at translate time rather than at runtime, and also provides the necessary information for the translator to coerce numeric argument types. This is reasonable because OPL does not support argument overloading. The same coercion occurs as when calling the built-in keywords.

Following the example of C and C++, you would normally provide a header file declaring prototypes of all the procedures and INCLUDE this header file at the beginning of the module that defines the declared procedures to ensure consistency. The header file would also be INCLUDEd in any other modules that call these procedures. Then you should use DECLARE EXTERNAL at the beginning of modules that include the header file so that the translator can ensure these procedures are called with correct parameter types, or types that can be coerced.

The following is an example of usage of DECLARE EXTERNAL and EXTERNAL:

```
DECLARE EXTERNAL
EXTERNAL myProc%:(i%,l&)
```

```
REM or INCLUDE "myproc.oph" that defines all your
    procedures

PROC test:
 LOCAL i%,j%,s$(10)

 REM j% is coerced to a long integer
 REM as specified by the prototype.
 myProc%:(i%,j%)

 REM translator 'Type mismatch' error:
 REM string can't be coerced to numeric type
 myProc%:(i%,s$)

 REM wrong argument count gives translator error
 myProc%:(i%)
ENDP

PROC myProc%:(i%,l&)
 REM Translator checks consistency with prototype
     above
 ...
ENDP
```

See DECLARE EXTERNAL.

FIRST — Positions to the first record

Usage: FIRST

Positions to the first record in the current view.

FIX$ — Converts a number to a string

Usage: f$=FIX$(x,y%,z%)

Returns a string representation of the number x, to y% decimal places. The string will be up to z% characters long.

Example: FIX$(123.456,2,7) returns "123.46".

If z% is negative then the string is right-justified, for example FIX$(1,2,-6) returns "1.00" where there are two spaces to the left of the 1.

If z% is positive then no spaces are added, for example FIX$(1,2,6) returns "1.00".

If the number x will not fit in the width specified by z%, then the string will just be asterisks, for example FIX$(256.99,2,4) returns "****".

See also GEN$, NUM$, SCI$.

FLAGS — Sets an application's system flags

Usage: FLAGS flags%

Used within an APP...ENDA construct to provide the OPL application's system flags. Possible values for flags% are:

KFlagsAppFileBased%	1	this application can create files. It will be included in the list of applications offered when the user creates a new file from the System screen
KFlagsAppIsHidden%	2	this application does not appear on the Extras bar. It is very unusual to have this flag set

These constants can be added together to combine their effects. They are supplied in Const.oph.

FLAGS may only be used within the APP...ENDA construct.

See also APP and OPL applications.

FLT — Converts an integer to a floating point number

Usage: f=FLT(x&)

Converts an integer expression (either integer or long integer) into a floating point number. Example:

```
PROC gamma:(v)
  LOCAL c
  c=3E8
  RETURN 1/SQR(1-(v*v)/(c*c))
ENDP
```

You could call this procedure like this: gamma:(FLT(a%)) if you wanted to pass it the value of an integer variable without having first to assign the integer value to a floating point variable.

See also INT and INTF.

FONT — Sets the text window's font and style

Usage: FONT id&,style%

Sets the text window's font and style. Font constants are provided in Const.oph, as they can be machine-dependent.

Standard font styles are:

KgStyleNormal%	0	normal style
KgStyleBold%	1	bold
KgStyleUnder%	2	underline
KgStyleInverse%	4	inverse video
KgStyleDoubleHeight%	8	double height
KgStyleMonoFont%	16	mono-spaced (typewriter) font
KgStyleItalic%	32	italic

All these constants are provided in Const.oph.

FREEALLOC Frees a previously allocated cell

Usage: FREEALLOC pcell&

Frees a previously allocated cell at pcell&.

See also SETFLAGS if you require the 64K limit to be enforced. If the flag is set to restrict the limit, pcell& is guaranteed to fit into a short integer.

gAT Sets the drawing position using absolute coordinates

Usage: gAT x%,y%

Sets the current position using absolute coordinates. gAT 0,0 moves to the top left of the current drawable.

See also gMOVE.

gBORDER Draws a border

Usage: gBORDER flags%,width%,height%
or gBORDER flags%

gBORDER is included for compatibility with older versions of OPL, however, it is recommended that programmers use gCREATE and gXBORDER in preference to this function (see below).

Draws a one-pixel wide, black border around the edge of the current drawable. If width% and height% are supplied, a border shape of this size is drawn with the top left corner at the current position. If they are not supplied, the border is drawn around the whole of the current drawable.

flags% controls three attributes of the border: a shadow to the right and beneath, a one-pixel gap all around, and the type of corners used. Its value can be built from:

KBordSglShadow%	1	single pixel shadow
KBordSglGap%	2	removes a single pixel shadow
KBordDblShadow%	3	double pixel shadow
KBordDblGap%	4	removes a double pixel shadow
KBordGapAllRound%	$100	one pixel gap all round
KBordRoundCorners%	$200	more rounded corners
KBordLosePixel%	$400	less rounded corners (only one pixel missing at the corner)

These constants are supplied in Const.oph.

These shadows do not appear in the same way that shadows on other objects, such as dialogs and menu panes, appear. To display such shadows on a window, you must specify them when using gCREATE. Hence you should use gCREATE (and gXBORDER) in preference to gBORDER.

You can combine the values to control the three different effects. (1, 2, 3, and 4 are mutually exclusive; you cannot use more than one of them.) For example, for rounded corners and a double pixel shadow, use flags%=$203.

Set flags%=0 for no shadow, no gap, and sharper corners.

For example, to de-emphasize a previously emphasized border, use gBORDER with the shadow turned off:

```
gBORDER 3     REM show border
GET
gBORDER 4     REM border off
...
```

See also gXBORDER.

gBOX — Draws a box

Usage: gBOX width%,height%

Draws a box from the current position, width% to the right and height% down. The current position is unaffected.

any of

```
gBUTTON text$,type%,w%,h%,state%
gBUTTON text$,type%,w%,h%,state%,bmpId&
gBUTTON text$,type%,w%,h%,state%,bmpId&,
    maskId&
gBUTTON text$,type%,w%,h%,state%,bmpId&,
    maskId&,layout%
```

Draws a 3D black and grey button at the current position in a rectangle of the supplied width w% and height h%, which fully encloses the button in all its states. text$ specifies up to 64 characters to be drawn in the button in the current font and style. You must ensure that the text will fit in the button.

The type% argument specifies the type of button to be drawn. For Symbian OS, this type% should be KButtS5%, although different values are supported by gBUTTON for backwards compatibility. Not all button states are supported by older button types.

KButtS5%	2	the standard Symbian OS button type. This is the style of button used on the 9210 and other devices

The state% argument gives the button state:

KButtS5Raised%	0	a raised button
KbuttS5SemiPressed%	1	a semi-depressed (flat) button
KbuttS5Sunken%	2	a fully-depressed (sunken) button

These constants are provided in Const.oph.

gCIRCLE Draws a circle

Usage: gCIRCLE radius%
or gCIRCLE radius%,fill%

Draws a circle with the center at the current position in the current drawable. If the value of radius% is negative then no circle is drawn.

If fill% is supplied and if fill%<>0 then the circle is filled with the current pen color.

See gELLIPSE, gCOLOR.

gCLOCK Draws or removes a clock

Usage: any of

```
gCLOCK ON/OFF
gCLOCK ON,mode%
gCLOCK ON,mode%,offset&
gCLOCK ON,mode%,offset&,format$
gCLOCK ON,mode%,offset&,format$,font&
gCLOCK ON,mode%,offset&,format$,font&,style%
```

Displays or removes a clock showing the system time. The current position in the current window is used. Only one clock may be displayed in each window.

mode% controls the type of clock:

KgClockS5System%	6	black and grey medium, system setting
KgClockS5Analog%	7	black and grey medium, analog
KgClockS5Digital%	8	second type medium, digital
KgClockS5LargeAnalog%	9	black and grey, extra large, analog
KgClockS5Formatted%	11	formatted digital

The digital clock (mode%=KgClockS5Digital%) automatically displays the day of the week and day of the month below the time. The extra large analog clock (mode%=KgClockS5LargeAnalog%) automatically displays a second hand.

Warning: Do not use gSCROLL to scroll the region containing a clock. When the time is updated, the old position would be used. The whole window may, however, be moved using gSETWIN.

Digital clocks display in 24-hour or 12-hour mode according to the system-wide setting.

offset& specifies an offset in minutes from the system time to the time displayed. This allows you to display a clock showing a time other than the system time. A flag that has the value $100 may be ORed with mode% so that offset& may be specified in seconds rather than minutes. The offset is a long integer to enable a whole day to be specified when the offset is in seconds.

If these arguments are not supplied, mode% is taken as 1 and offset& as 0.

The system setting for the clock type (i.e. digital or analog) can be changed by an OPL program using the procedure LCSETCLOCKFORMAT: in the Date OPX

(see Date.opx – Date and time manipulation). This function should be used to implement, for example, tapping a toolbar clock to change its type.

format$, font%, and style% are used only for formatted digital clocks (mode% 11 on EPOC). The values for font& and style% are as for gFONT and gSTYLE. The default font for gCLOCK is the system font. The default style is normal (0).

For the formatted digital clock, a format string (up to 255 characters long) specifies how the clock is to be displayed. The format string contains a number of format specifications in the form of a % followed by a letter. Uppercase or lowercase may be used.

The format string may contain the following symbols to obtain the required effects:

%%
Insert a single % character in the string

%*
Abbreviate following item. (The asterisk should be inserted between the % and the number or letter, e.g. %*1.) In most cases this amounts to omitting any leading zeros, for example, if it is the first of the month "%F %*M" will display as 1 rather than 01.

%:n
Insert a system time separator character. n is an integer between zero and three inclusive, indicating which time separator character is to be used. For European time settings, only n=1 and n=2 are used, giving the hours/minutes separator and minutes/seconds separator, respectively.

%/n
Insert a system date separator character. n is an integer between zero and three inclusive, indicating which date separator character is to be used. For European time settings, only n=1 and n=2 are used, giving the day/month separator and month/year separator, respectively.

%1
Insert the first component of a three-component date (i.e. a date including day, month, and year) where the order of the components is determined by the system settings. The possibilities are: dd/mm/yyyy (European), mm/dd/yyyy (American), yyyy/mm/dd (Japanese).

%2
Insert the second component of a three-component date, where the order has been determined by the system settings. See %1.

%3
Insert the third component of a three-component date, where the order has been determined by the system settings. See %1.

%4
Insert the first component of a two-component date (i.e. a date including day and month only), where the order has been determined by system settings. The possibilities are: dd/mm (European), mm/dd (American), mm/dd (Japanese).

%5
Insert the second component of a two-component date, where the order has been determined by the system settings. See %4.

%A
Insert a.m. or p.m. according to the current language and time of day. Text is printed even if 24-hour clock is in use. Text may be specified to be printed before or after the time, and a trailing or leading space as appropriate will be added. The abbreviated version (%*A) removes this space.

Optionally, a minus or plus sign may be inserted between the % and the A. This operates as follows: %−A causes a.m./p.m. text to be inserted only if the system setting of the a.m./p.m. symbol position is set to display before the time. Similarly, %+A causes a.m./p.m. text to be inserted only if the system setting of the a.m./p.m. symbol is set to display after the time. No a.m./p.m. text will be inserted before the time if a + is inserted in the string. For example, you could use "%−A%H%:1%T%+A" to insert the a.m./p.m. symbol either before or after the time, according to the system setting. %+A and %−A cannot be abbreviated.

%B
As %A, except that the a.m./p.m. text is only inserted if the system clock setting is 12 hour. (This should be used in conjunction with %J.)

%D
Insert the two-digit day number in month (in conjunction with %1, etc.).

%E
Insert the day name. Abbreviation is language-specific (3 letters in English).

%F
Use this at the beginning of a format string to make the date/time formatting independent of the system setting. This fixes the order of the following day/month/year component(s) in their given order, removing the need to use %1 to %5, allowing individual components of the date to be printed. (No abbreviation.)

%H
Insert the two-digit hour component of the time in 24-hour clock format.

%I
Insert the two-digit hour component of the time in 12-hour clock format. Any leading zero is automatically suppressed, regardless of whether an asterisk is inserted or not.

%J
Insert the two-digit hour component of time in either 12- or 24-hour clock format depending on the corresponding system setting. When the clock has been set to 12-hour format, the hour's leading zero is automatically suppressed, regardless of whether an asterisk has been inserted between the % and J.

%M
Insert the two-digit month number (in conjunction with %1, etc.).

%N
Insert the month name (in conjunction with %1, etc.). When using system settings (i.e. not using %F) this causes all months following %N in the string to be written in words. When using fixed format (i.e. when using %F), %N may be used alone to insert a month name. Abbreviation is language-specific (3 letters in English).

%S
Insert the two-digit second component of the time.

%T
Insert the two-digit minute component of the time.

%W
Insert the two-digit week number in year, counting the first (part) week as week 1.

%X
Insert the date suffix. When using system settings (i.e. not using %F), this causes a suffix to be put on any date following %X in the string. When using fixed format (i.e. using %F), %X following any date appends a suffix for that particular date. Cannot be abbreviated.

%Y
Insert the four-digit year number (in conjunction with %1, etc.). The abbreviation is the last two digits of the year.

%Z
Insert the three-digit day number in year.

Some examples of the use of these format strings are as follows. The example use is 1:30:05 p.m. on Wednesday, 1st January 1997, with the system setting of European dates and with a.m./p.m. after the time:

"%-A%I:%T:%S%+A" will print the time in 12-hour clock, including seconds, with the a.m./p.m. either inserted before or after the time, depending on the system setting. So the example time would appear as 1:30:05 p.m.

"%F%E %*D%X %N %Y" will print the day of the week followed by the date with suffix, the month as a word, and the year. For example, Wednesday 1st January 1997.

"%E %D%X%N%Y %1%2%3" will use the locale setting for ordering the elements of the date, but will use a suffix on the day and the month in words. For example, Wednesday 01st January 1997.

"%*E %*D%X%*N%*Y %1 %2 '%3" will be similar to 3, but will abbreviate the day of the week, the day, the month, and the year, so the example becomes "Wed 1st Jan 97".

"%M%Y%D%1%/0%2%/0%3" will appear as 01/01/1997. This demonstrates that the ordering of the %D, %M, and %Y is irrelevant when using locale-dependent formatting. Instead the ordering of the date components is determined by the order of the %1, %2, and %3 formatting commands.

style% may take any of the values used to specify gSTYLE, other than 2 (underlined).

A note should also be made that a 'General failure' error will result if you attempt to use an invalid format. Invalid formats include using %: and %/ followed by 0 or 3 when in European locale setting (when these separators are without meaning) and using %+ and %− followed by characters other than A or B.

gCLOSE Closes a drawable

Usage: gCLOSE id%

Closes the specified drawable that was previously opened by gCREATE, gCREATEBIT, or gLOADBIT.

If the drawable closed was the current drawable, the default window (ID=1) becomes current.

An error is raised if you try to close the default window.

gCLS Clears the current drawable

Usage: gCLS

Clears the whole of the current drawable and sets the current position to 0,0, its top left corner.

gCOLOR Sets the pen color

Usage: gCOLOR red%,green%,blue%

Sets the pen color of the current window. The red%, green%,blue% values specify a color that will be mapped to white, black, or one of the greys on non-color screens. Note that if the values of red%, green%, and blue% are equal, then a pure grey results, ranging from black (0) to white (255).

See also gCOLORBACKGROUND, gCOLORINFO.

gCOLORBACKGROUND Sets the background color

Usage: gCOLORBACKGROUND red%, green%,blue%

Sets the background, or 'paper' color of the current graphics window. Subsequent graphics commands in the window will use this background color. The red%,green%,blue% values specify a color that will be mapped to

white, black, or one of the greys on non-color screens. Note that if the values of red%, green%, and blue% are equal, then a pure grey results, ranging from black (0) to white (255).

For example:

```
gUSE 1
gCOLOR $ff,0,0
gCOLORBACKGROUND 0,0,$ff
gAT 20,20
gPRINTB "Red text on a blue
    background",250
```

See also gCOLORINFO, gCOLOR.

gCOLORINFO Gets the Symbian OS phone's color information

Usage: gCOLORINFO var cinfo&()

Interrogates the system to find the maximum number of colors available on the Symbian OS phone. cinfo&() must be at least seven elements long, and on gCOLORINFO's return will contain information indexed by the following values:

gColorInfoADisplayMode%	1	default window mode
gColorInfoANumColors%	2	number of colors supported
gColorInfoANumGreys%	3	number of greys supported

The remaining four elements are reserved for future use.

The default window mode will be one of the following values:

KDisplayModeNone%	0	
KDisplayModeGray2%	1	2 greys
KDisplayModeGray4%	2	4 greys
KDisplayModeGray16%	3	16 greys
KDisplayModeGray256%	4	256 greys
KDisplayModeColor16%	5	16 colors
KDisplayModeColor256%	6	256 colors
KDisplayModeColor64K%	7	65,536 colors (16-bit color)
KDisplayModeColor16M%	8	16,777,216 colors (24-bit color)
KDisplayModeRGB%	9	
KDisplayModeColor4K%	10	4096 colors (12-bit color)

For example, to find out about the color depth of the display on your device:

```
include "Const.oph"
proc ColTest:
  local c&(3)
  gColorInfo c&()
  print c&(gColorInfoADisplayMode%),
  print c&(gColorInfoANumColors%),
  print c&(gColorInfoANumGreys%)
  get
endp
```

See also gCOLORBACKGROUND, gCOLOR.

gCOPY Copies a rectangular area

Usage: gCOPY id%,x%,y%,w%,h%,mode%

Copies a rectangle of the specified size (width w%, height h%) from the point x%,y% in drawable id%, to the current position in the current drawable.

It is inadvisable to use gCOPY to copy from windows as it is very slow. It should only be used for copying from bitmaps to windows or other bitmaps.

As this command can copy both set and clear pixels, the same modes are available as when displaying text. Possible values for mode% are:

KtModeSet%	0	Set
KtModeClear%	1	Clear
KtModeInvert%	2	Invert
KtModeReplace%	3	Replace

Set, Clear, and Invert act only on set pixels in the pattern; Replace copies the entire rectangle, with set and clear pixels.

The current position is not affected in either window.

gCREATE Creates a window

Usage: either of

```
id%=gCREATE(x%,y%,w%,h%,v%)

id%=gCREATE(x%,y%,w%,h%,v%,flags%)
```

Creates a window with specified position and size (width w%, height h%), and makes it both current and foreground. The current position is set initially to (0,0), the top left corner.

The v% argument specifies whether the window is initially visible:

KgCreateInvisible%	0	invisible window
KgCreateVisible%	1	visible window

gCREATE returns the ID of the window. Window IDs are used as arguments to other functions and keywords, and refer to the newly created window.

Window mode flags

The flags% argument specifies the graphics mode and shadowing style to use on the window. The defaults are 2-color, with no shadow.

It is simplest to specify flags% as a hexadecimal number, as its value is a bitwise OR of significant values.

The least significant 4 bits of flags% (masked by $000F) give the color mode. The next 4 bits (masked by $00F0) specify the window's shadowing. The following constant values can be added, or ORed to give the first 8 bits of flags%:

KColorgCreate2GrayMode%	$0000	2-grey mode
KColorgCreate4GrayMode%	$0001	
KColorgCreate16GrayMode%	$0002	
KColorgCreate256GrayMode%	$0003	256-grey mode
KColorgCreate16ColorMode%	$0004	16-color mode
KColorgCreate256ColorMode%	$0005	256-color mode
KgCreateHasShadow%	$0010	window has a shadow

These constants are provided in Const.oph. The old constants, KgCreate2ColorMode%, KgCreate4ColorMode%, and KgCreate16ColorMode%, are still retained for backwards compatibility.

Note: It is not an error to create a window with features not supported on the hardware, but it is inefficient and should be avoided. This is similar to the concept of copying a 4-grey bitmap into a 2-grey window on greyscale machines: processor time is wasted dithering the colors down to the same level as the window.

Bits 8–11 (masked by $0F00) give the shadow height relative to the window behind it as a left-shifted 4-bit number. A height of N units gives a shadow of N*2 pixels.

Examples:

flags%	description
$412	16-color mode ($2), shadowed window ($1), with height 4 units ($4) above the previous window with a shadow of 8 pixels
$010	2-color mode (black and white) shadowed window at the same height as the previous window
$101	4-color mode window with no shadow (height ignored if shadow disabled)
$111	4-color mode window with shadow of 1 unit above window behind, i.e. 2 pixel shadow

Note that 63 windows may be open at any time and it is recommended that you use many small windows rather than a few large ones.

See also gCLOSE, gCOLOR, DEFAULTWIN.

gCREATEBIT Creates a bitmap

Usage:

```
id%=gCREATEBIT(w%,h%)

id%=gCREATEBIT(w%,h%,mode%)
```

Creates a bitmap with the specified width and height, and makes it the current drawable. Sets the current position to 0,0, its top left corner.

Returns id%, which identifies this bitmap for other keywords.

gCREATEBIT may be used with an optional third parameter that specifies the graphics mode of the bitmap to be created. The values of these are as given in gCREATE. By default the graphics mode of a bitmap is 2-color.

See also gCLOSE, gCREATE.

gELLIPSE Draws an ellipse

Usage: gELLIPSE hRadius%,vRadius%
or gELLIPSE hRadius%,vRadius%,fill%

Draws an ellipse with the center at the current position in the current drawable. hRadius% is the horizontal distance in pixels from the center of the ellipse to the

left (and right) of the ellipse. vRadius% is the vertical distance from the center of the ellipse to the top (and bottom). If the length of either radius is less than zero, then no ellipse is drawn.

If fill% is supplied and if fill%<>0 then the ellipse is filled with the current pen color.

See gCIRCLE, gCOLOR.

GEN$ Converts a number to a string

Usage: g$=gen$(x,y%)

Returns a string representation of the number x. The string will be up to y% characters long.

Example: GEN$(123.456,7) returns "123.456" and GEN$(243,5) returns "243".

If y% is negative then the string is right-justified – for example, GEN$(1,-6) returns "1" where there are five spaces to the left of the 1.

If y% is positive then no spaces are added, for example GEN$(1,6) returns "1".

If the number x will not fit in the width specified by y%, then the string will just be asterisks, for example GEN$(256.99,4) returns "****".

See also FIX$, NUM$, SCI$.

GET Waits for and returns the keycode for the next key pressed

Usage: g%=GET

Waits for a key to be pressed and returns the character code for that key.

For example, if the A key is pressed with Caps Lock off, the integer returned is 97 (a), or 65 (A) if A was pressed with the Shift key down.

The character codes of special keys, such as Pg Dn, are given in Appendix D.

You can use KMOD to check whether modifier keys (Shift, Ctrl, Fn, and Caps) were used.

See also KEY.

GET$ Waits for and returns the next key pressed as a string

Usage: g$=GET$

Waits until a key is pressed and then returns which key was pressed, as a string.

For example, if the A key is pressed in lowercase mode, the string returned is "a".

You can use KMOD to check whether any modifier keys (Shift, Ctrl, Fn, and Caps) were used.

See also KEY$.

GETCMD$ Gets new command line arguments

Usage: w$=GETCMD$

Returns new command line arguments to an OPL application, after a change files or quit event has occurred. Usually this is called after GETEVENT32 has returned a system command.

The command line arguments are returned as a string. The first character of the return value has the following meaning:

KGetCmdLetterCreate$	"C"	close down the current file, and create the specified new file
KGetCmdLetterOpen$	"O"	close down the current file, and open the specified existing file
KGetCmdLetterExit$	"X"	close down the current file (if any) and quit the app

These constants are defined in Const.oph.

If the first character is KGetCmdLetterCreate$ or KGetCmdLetterOpen$, then the rest of the returned string is a filename.

You can only call GETCMD$ once for each system message.

GETDOC$ Gets the name of the current document

Usage: docname$=GETDOC$

Returns the name of the current document.

See also SETDOC.

GETEVENT Deprecated version of GETEVENT32

This keyword is included for compatibility with older versions of the OPL language.

It is strongly recommended that you use GETEVENT32 rather than GETEVENT. GETEVENT is supported only for backward compatibility and cannot be used to handle pen events in a satisfactory way.

GETEVENT32 Synchronous wait for event

Section contents
Key press events
Foreground, background, and switch on events
Command events (exit or file switch)
Key down and key up events
Pen point events
Event notes

Usage: GETEVENT32 ev&()

Waits for an event to occur, and returns with ev&() specifying the event. The data returned in ev&() depends on the type of event that occurred.

All events return a 32-bit time stamp. The window ID mentioned below refers to the value returned by the gCREATE keyword. The modifier values and scancode values for a key press (which specify a location on the keyboard) are given in the Character codes.

Const.oph supplies the following constants for indexing ev&():

KEvAType%	1	type of the event
KEvATime%	2	32-bit time stamp

These event index numbers and their meanings are the same for all possible kinds of event. ev&(KEvAType%) is always the event type, and ev&(KEvATime%) is always the time stamp for the event. The meanings of other values in ev&() are dependent on the type of event.

Keypress events

Note that keycodes are returned in the first element of ev&(), ev&(KEvAType%). To recognize a key press event, you have to use a bit mask. Key press events are masked by KEvNotKeyMask&.

For a key press event, (ev&(KEvAType%) AND KEvNotKeyMask&) is always 0.

KEvNotKeyMask&	&400	masks out non-key press events

The index values for a key event are:

–	1	key code (cooked)
–	3	scan code (raw)
KEvAKMod%	4	index for the key modifier
KEvAKRep%	5	index for the repeat value

ev&(KEvATime%) is the time stamp, as with any other event.

Foreground, background, and switch on events

For these events, ev&(KEvAType%) is one of:

KEvFocusGained&	&401	program has moved to foreground
KEvFocusLost&	&402	program has moved to background
KEvSwitchOn&	&403	machine is switched on

Note that KEvSwitchOn& is only returned by GETEVENT32 if the appropriate flag is set by a call to SETFLAGS. GETEVENT32 ignores the machine being switched on if the flag is not set.

Command events (exit or file switch)

This kind of event should be handled specially by programmers. For this event, ev&(KEvAType%) is

KEvCommand&	&404	a command

A command is passed to an application when the Operating System wants the application to switch files or exit. If this event is received, GETCMD$ should be called to find out what action should be taken.

Key down and key up events

For these events, ev&(KEvAType%) is either of:

KEvKeyDown&	&406	key pressed down
KEvKeyUp&	&407	key released

It is only possible to extract the scancode of a key up or key down event. However, for each user key press, three events are generated: key up, key down, and key press. The keycode can be extracted from the key press event (see above).

ev&() indices for these event types are:

–	3	key down or key up event's scancode
–	4	key modifiers

Pen point events

For pen events, ev&(KEvAType%) is one of:

KEvPtr&	&408	pen event
KEvPtrEnter&	&409	pen point enters a window
KEvPtrExit&	&40A	pen point leaves a window

The ev&() for a pointer event is a nine-element array indexed by the following values:

KEvAPtrOplWindowId%	3	ID of parent window
KEvAPtrWindowId%	3	synonym for KEvAPtrOplWindowId%
KEvAPtrType%	4	type of pointer event (see below)
KEvAPtrModifiers%	5	modifiers
KEvAPtrPositionX%	6	X coordinate
KEvAPtrPositionY%	7	Y coordinate
KEvAPtrScreenPosX%	8	X coordinate relative to the parent window
KEvAPtrScreenPosY%	9	Y coordinate relative to the parent window

ev&(KEvAPtrType%) contains the type of pointer event. A given device may not support all types of pointer event, because of the wide range of possible pointer input devices. ev&(KEvAPtrType%) will contain one of the following values:

KEvPtrPenDown&	0	pen down
KEvPtrPenUp&	1	pen up
KEvPtrButton1Down&	0	button 1 down
KEvPtrButton1Up&	1	button 1 up
KEvPtrButton2Down&	2	button 2 down
KEvPtrButton2Up&	3	button 2 up
KEvPtrButton3Down&	4	button 3 down
KEvPtrButton3Up&	5	button 3 up
KEvPtrDrag&	6	pointer drag (while button held down)
KEvPtrMove&	7	pointer move (without any button being pressed)
KEvPtrButtonRepeat&	8	button repeat
KEvPtrSwitchOn&	9	machine turned on by a screen touch

Event notes

Constants for event codes and subscripts are supplied in Const.oph.

Some pointer events, and pointer enters and exits, can be filtered out to avoid being swamped by unwanted event types. See POINTERFILTER.

For unknown events, ev&(1) contains &1400 added to the code returned by the window server. ev&(2) is the timestamp, ev&(3) is the window ID, and the rest of the data returned by the window server is put into ev&(4), ev&(5), etc.

If a non-key event such as 'foreground' occurs while a keyboard keyword such as GET, INPUT, MENU, or DIALOG is being used, the event is discarded. So GETEVENT must be used if non-key events are to be monitored. If you need to use these keywords in applications, use LOCK ON/LOCK OFF around them, so that the System screen won't send messages to switch files or shutdown while the application cannot respond.

See also GETEVENTA32.

GETEVENTA32 Waits for an event asynchronously

Usage: GETEVENTA32 status%,ev&()

Asynchronous version of GETEVENT32. GETEVENTA32 returns the same codes to the array ev&() as GETEVENT32.

The exist status of GETEVENTA32 is placed into status% when an event is received. For this reason, status% should probably be a global variable.

See GETEVENTC, GETEVENT32, GETEVENT.

GETEVENTC Cancels an outstanding GETEVENT32

Usage: GETEVENTC(var stat%)

Cancels the previously called GETEVENTA32 function with status stat%. Note that GETEVENTC consumes the signal (unlike IOCANCEL), so IOWAITSTAT should not be used after GETEVENTC.

gFILL Draws a filled rectangle

Usage: gFILL width%,height%,gMode%

Fills a rectangle of the specified size from the current position, according to the graphics mode specified.

The current position is unaffected.

gFONT Sets the current drawable's font

Usage: gFONT fontUid&

Sets the font for current drawable to fontId&. The font may be one of the predefined fonts in the ROM or a user-defined font. See Graphics for more details of fonts.

Constants for the font UIDs are supplied in Const.oph.

User-defined fonts must first be loaded by gLOADFONT, then the font UIDs of the loaded fonts may be used with gFONT. Note that this is not the ID returned by gLOADFONT (which is the font file ID), but the UID defined in the font file itself.

See also gLOADFONT, FONT.

gGMODE Sets the effect of subsequent drawing commands

Usage: gGMODE mode%

Sets the effect of all subsequent drawing commands gLINEBY, gBOX, etc. on the current drawable.

KgModeSet%	0	pixels will be set
KgModeClear%	1	pixels will be cleared
KgModeInvert%	2	pixels will be inverted

These constants are supplied in Const.oph.

When you first use drawing commands on a drawable, they set pixels in the drawable. Use gGMODE to change this. For example, if you have drawn a black background, you can draw a white box outline inside it with either gGMODE 1 or gGMODE 2, followed by gBOX.

gGREY Changes pen color between black and grey

Usage: gGREY mode%

Changes the pen color between black and grey. mode% has the following effects:

mode%=1 sets the foreground color of the current drawable to light grey. This is the same color as would be achieved by using gCOLOR $aa,$aa,$aa.

mode% of any other value sets the foreground color to black (the default).

See also DEFAULTWIN and gCREATE.

gHEIGHT Height of the current drawable

Usage: height%=gHEIGHT

Returns the height of the current drawable.

gIDENTITY ID of the current drawable

Usage: id%=gIDENTITY

Returns the ID of the current drawable.
The default window has ID=1.

gINFO32 Gets information about the current drawable

Usage: gINFO32 var i&()

Gets general information about the current drawable and about the graphics cursor (whichever window it is in) changed. The information is returned in the array i%(), which must be at least 32 integers long. i&() must have 48 elements, although elements 37 to 48 are currently unused.

i%(1)	reserved
i%(2)	reserved
i%(3)	height of font
i%(4)	descent of font
i%(5)	ascent of font
i%(6)	width of '0' character
i%(7)	maximum character width
i%(8)	flags for font (see below)
i%(9)	the font UID as used in gFONT
i%(10–17)	unused
i%(18)	current graphics mode (gGMODE)
i%(19)	current text mode (gTMODE)
i%(20)	current style (gSTYLE)
i%(21)	cursor state (ON=1,OFF=0)
i%(22)	ID of window containing cursor (–1 for text cursor)
i%(23)	cursor width
i%(24)	cursor height
i%(25)	cursor ascent
i%(26)	cursor x position in window
i%(27)	cursor y position in window
i%(28)	1 if drawable is a bitmap
i%(29)	cursor effects
i%(30)	graphics color mode of the current window. This will be one of the values described in DEFAULTWIN
i%(31)	gCOLOR red% of foreground
i%(32)	gCOLOR green% of foreground
i%(33)	gCOLOR blue% of foreground
i&(34)	gCOLOR red% of background
i&(35)	gCOLOR green% of background
i&(36)	gCOLOR blue% of background

The value given for i%(8) is a combination of the following values:

1	font uses standard ASCII characters (32–126)
2	font uses Code Page 1252 characters (128–255)
4	font is bold
8	font is italic
16	font has serifs
32	font is monospaced
$8000	font is stored expanded for quick drawing

i%(29) has bit 1 set (i%(29) AND 2) if the cursor is not flashing, and bit 2 set (i%(29) AND 4) if it is grey.

If the cursor is off (i%(21)=0) or is a text cursor (i%(22)=-1), i%(23) to i%(27) and i%(29) should be ignored.

See also gFONT, gCOLOR, gCREATE.

gINVERT Draws an inverted rectangle

Usage: gINVERT width%,height%

Inverts the rectangle width% to the right and height% down from the cursor position, except for the four corner pixels.

GIPRINT Displays an information message

Usage: GIPRINT str$,c%
or GIPRINT str$

Displays an information message for about two seconds, in the bottom right corner of the screen. For example, GIPRINT "Not Found" displays Not Found. If a string is too long for the screen, it will be clipped.

If c% is given, it controls the corner in which the message appears. Corner constants for the information message are as given for BUSY.

Only one message can be shown at a time. You can make the message go away – for example, if a key has been pressed – with GIPRINT "".

gLINEBY Draws a line

Usage: gLINEBY dx%,dy%

Draws a line from the current position to a point dx% to the right and dy% down from the starting point. Negative dx% and dy% mean left and up, respectively. The end point is not drawn, so for gLINEBY dx%,dy%, point gX+dx%,gY+dy% is not drawn. Note, however, that OPL specially plots the point when the start and end point coincide.

The current position moves to the end of the line drawn. gLINEBY 0,0 sets the pixel at the current position.

See also gLINETO, gPOLY.

gLINETO Draws a line to an absolute position

Usage: gLINETO x%,y%

Draws a line from the current position to the point x%,y%. The current position moves to x%,y%. The end point is not drawn, so for gLINETO x%,y%, point x%,y% is not drawn. Note, however, that OPL specially plots the point when the start and end point coincide.

To plot a single point on all machines, use gLINETO to the current position (or gLINEBY 0,0).

See also gLINEBY, gPOLY.

gLOADBIT Loads a bitmap

Usage: any of

```
id%=gLOADBIT(name$,write%,index%)
id%=gLOADBIT(name$,write%)
id%=gLOADBIT(name$)
```

Loads a bitmap from the named bitmap file and makes it the current drawable. Sets the current position to 0,0, its top left corner.

gLOADBIT loads Symbian OS Picture files, which are naturally in the same file format that is saved by gSAVEBIT. Symbian OS Picture files can also be generated by exporting files created by the Sketch application. These are called multi-bitmap files (MBMs), though often containing just one bitmap as in the case of gSAVEBIT or Sketch files, and are often given an extension .MBM.

The bitmap is kept as a local copy in memory.

gLOADBIT returns id%, which identifies this bitmap for other keywords.

A write% argument of KgLoadBitReadOnly% sets read-only access. Attempts to write to the bitmap in memory will be ignored, but the bitmap can be used by other programs without using more memory. Using KgLoadBitWriteable% for write% allows you to write to and re-save the bitmap, which is the default.

KgLoadBitReadOnly%	0	read-only access
KgLoadBitWriteable%	1	read and write access

Constants are supplied in Const.oph.

For bitmap files that contain more than one bitmap, index% specifies which one to load. For the first bitmap, use index%=0, which is also the default value.

See also gCLOSE.

gLOADFONT Loads a font

Usage: fileId%=gLOADFONT(file$)

Loads the user-defined fonts specified in the file file$ and returns the file ID of the font file, which can be used only with gUNLOADFONT. The maximum number of font files that may be loaded at any one time is 16.

To use the fonts in a loaded font file you need to use their published UIDs, which will be defined in the font file itself, for example:

```
fileId%=gLOADFONT("Music1")
gFONT KMusic1Font1&
...
gUNLOADFONT fileId%
```

gFONT itself is very efficient, so you should normally load all required fonts at the start of a program.

Note that the built-in fonts are automatically available, and do not need loading.

See gUNLOADFONT.

GLOBAL Declares global variables

Usage: GLOBAL variables

Declares variables to be used in the current procedure (as does the LOCAL command) and (unlike LOCAL) in any procedures called by the current procedure, or procedures called by them.

The variables may be of four types, depending on the symbol they end with:

Variable names not ending with $, %, &, or () are floating point variables, for example price, x
Those ending with a % are integer variables, for example x%, sales92%
Those ending with an & are long integer variables, for example x&, sales92&
Those ending with a $ are string variables. String variable names must be followed by the maximum length of the string in brackets, for example names$(12), a$(3)

Array variables have a number immediately following them in brackets that specifies the number of elements in the array. Array variables may be of any type, for example: x(6),y%(5),f$(5,12),z&(3).

When declaring string arrays, you must give two numbers in the brackets. The first declares the number of elements, the second declares their maximum length. For example, surname$(5,8) declares five elements, each up to eight characters long.

Variable names may be any combination of up to 32 numbers and alphabetic characters and the underscore character. They must start with an alphabetic character or an underscore.

The length of a variable name includes the %, &, or $ sign, but not the () in string and array variables.

More than one GLOBAL or LOCAL statement may be used, but they must be on separate lines, immediately after the procedure name.

See also LOCAL.

gMOVE Moves the current position

Usage: gMOVE dx%,dy%

Moves the current position dx% to the right and dy% downwards, in the current drawable.

A negative dx% causes movement to the left; a negative dy% causes upward movement.

See also gAT.

gORDER Moves the selected window

Usage: gORDER id%,position%

Sets the window specified by id% to the selected foreground/background position, and redraws the screen. Position 1 is the foreground window, position 2 is next, and so on. Any position greater than the number of windows is interpreted as the end of the list.

On creation, a window is at position 1 in the list.

Raises an error if id% is a bitmap.

See also gRANK.

gORIGINX Window's X position

Usage: x%=gORIGINX

Returns the gap between the left side of the screen and the left side of the current window.

Raises an error if the current drawable is a bitmap.

gORIGINY Window's Y position

Usage: y%=gORIGINY

Returns the gap between the top of the screen and the top of the current window.

Raises an error if the current drawable is a bitmap.

GOTO Jumps to a labeled line

Usage:

```
GOTO label or GOTO label::
...
label::
```

Goes to the line following the label:: and continues from there. The label:

Must be in the current procedure

Must start with a letter and end with a double colon, although the double colon is not necessary in the GOTO statement

May be up to 32 characters long, excluding the colons

GOTOMARK Makes a bookmarked record the current record

Usage: GOTOMARK b%

Makes the record with bookmark b%, as returned by BOOKMARK, the current record. b% must be a bookmark in the current view.

gPATT Draws a pattern-filled rectangle

Usage: gPATT id%,width%,height%,mode%

Fills a rectangle of the specified size from the current position with repetitions of the drawable id%.

As with gCOPY, this command can copy both set and clear pixels, so the same modes are available as when displaying text. Set mode%=0 for set, 1 for clear, 2 for invert, or 3 for replace. 0, 1, and 2 act only on set pixels in the pattern; 3 copies the entire rectangle, with set and clear pixels.

If you set id%=−1 a predefined grey pattern is used.

The current position is unaffected.

gPEEKLINE Reads a horizontal line from a drawable

Usage: gPEEKLINE id%,x%,y%,d%(),ln%,mode%

Reads a horizontal line from the black plane of the drawable id%, length ln%, starting at x%,y%. The leftmost 16 pixels are read into d%(1), with the first pixel read into the least significant bit.

gPEEKLINE has an extra optional parameter mode% to specify the color mode:

mode%	color mode	color of pixel that sets bits
−1	black and white	black
0	black and white	white
1	4-color mode	white
2	16-color mode	white

The default mode% is −1. For 4- and 16-color modes, 2 and 4 bits per pixel, respectively are used. This is to enable the color of the pixel to be ascertained from the bits that are set. White results in all 2 or 4 bits being set, while black sets none of them. For example, in a 4-color window, with the color set by

```
gCOLOR 16,16,16
```

a pixel of a line would peek as 0001 in binary. Similarly, a pixel of a line with the color set to

```
gCOLOR 80,80,80
```

would result in the value 0101 in binary when peeked.

The array d%() must be long enough to hold the data. You can work out the number of integers required with ((ln%+15)/16) (using whole-number division).

If the optional parameter mode% is used, the array size allowed must be adjusted accordingly: it must be at least twice as long as the array needed for black and white if the line you wish to peek is in 4-color mode, and four times as long in 16-color mode.

gPOLY — Draws a polygon

Usage: gPOLY a%()

Draws a sequence of lines, as if by gLINEBY and gMOVE commands.

The array is set up as follows:

a%(1)	starting x position
a%(2)	starting y position
a%(3)	number of pairs of offsets
a%(4)	dx1%
a%(5)	dy1%
a%(6)	dx2%
a%(7)	dy2%, etc.
...	}
...	} further pairs...

The following constants can be used for the first five array indices:

KgPolyAStartX%	1
KgPolyAStartY%	2
KgPolyANumPairs%	3
KgPolyANumDx1%	4
KgPolyANumDy1%	5

Each pair of numbers dx1%,dy1%, for example, specifies a line or a movement. To draw a line, dy% is the

amount to move down, while dx% is the amount to move to the right multiplied by two.

To specify a movement (i.e. without drawing a line) work out the dx%,dy% as for a line, then add 1 to dx%.

(For drawing/movement up or left, use negative numbers.)

gPOLY is quicker than combinations of gAT, gLINEBY, and gMOVE.

Example (to draw three horizontal lines 50 pixels long at positions 20,10, 20,30, and 20,50):

```
a%(1)=20 :a%(2)=10      REM 20,10
a%(3)=5                 REM 5 operations
a%(4)=50*2 :a%(5)=0     REM draw right 50
a%(6)=0*2+1 :a%(7)=20   REM move down 20
a%(8)=-50*2 :a%(9)=0    REM draw left 50
a%(10)=0*2+1 :a%(11)=20 REM draw left 50
a%(12)=50*2 :a%(13)=0   REM draw right 50
gPOLY a%()
```

gPRINT — Prints a list into a drawable

Usage: gPRINT list of expressions

Displays a list of expressions at the current position in the current drawable. All variable types are formatted as for PRINT.

Unlike PRINT, gPRINT does not end by moving to a new line. A comma between expressions is still displayed as a space, but a semicolon has no effect. gPRINT without a list of expressions does nothing.

See also gPRINTB, gPRINTCLIP, gTWIDTH, gXPRINT, gTMODE.

gPRINTB — Prints text into a cleared box

Usage: any of

```
gPRINTB t$,w%,al%,tp%,bt%,m%
gPRINTB t$,w%,al%,tp%,bt%
gPRINTB t$,w%,al%,tp%
gPRINTB t$,w%,al%
gPRINTB t$,w%
```

Displays text t$ in a cleared box of width w% pixels. The current position is used for the left side of the box and for the baseline of the text.

al% controls the alignment of the text in the box:

KgPrintBRightAligned%	1	right alignment
KgPrintBLeftAligned%	2	left alignment
KgPrintBDefAligned%	2	default alignment (left)
KgPrintBCentredAligned%	3	centered

tp% and bt% are the clearances between the text and the top/bottom of the box. Together with the current font size, they control the height of the box. An error is raised if tp% plus the font ascent is greater than 255.

m% controls the margins. For left alignment, m% is an offset from the left of the box to the start of the text. For right alignment, m% is an offset from the right of the box to the end of the text. For centering, m% is an offset from the left or right of the box to the region in which to center, with positive m% meaning left and negative meaning right:

KgPrintBDefTop%	0	default top clearance
KgPrintBDefBottom%	0	default bottom clearance
KgPrintBDefMargin%	0	default margin

These constants are supplied in Const.oph.

See also gPRINT, gPRINTCLIP, gTWIDTH, gXPRINT.

gPRINTCLIP Prints text that fits within defined area

Usage: w%=gPRINTCLIP(text$,width%)

Displays text$ at the current position, displaying only as many characters as will fit inside width% pixels. Returns the number of characters displayed.

See also gPRINT, gPRINTB, gTWIDTH, gXPRINT, gTMODE.

gRANK Gets foreground/background position of the current window

Usage: rank%=gRANK

Returns the foreground/background position – from 1 to 64 – of the current window. Raises an error if the current drawable is a bitmap.

See also gORDER.

gSAVEBIT Saves the current drawable

Usage: gSAVEBIT name$,width%,height%
or gSAVEBIT name$

Saves the current drawable as the named bitmap file. If width% and height% are given, then only the rectangle of that size from the current position is copied.

gSAVEBIT does not add a default filename extension to the input argument name if none is provided on the machine.

gSCROLL Scrolls pixels

Usage: gSCROLL dx%,dy%,x%,y%,wd%,ht%
or gSCROLL dx%,dy%

Scrolls pixels in the current drawable by offset dx%, dy%. Positive dx% means to the right, and positive dy% means down. The drawable itself does not change its position.

If you specify a rectangle in the current drawable, at x%,y% and of size wd%,ht%, only this rectangle is scrolled.

The areas dx% wide and dy% deep 'left behind' by the scroll are cleared.

The current position is not affected.

gSETPENWIDTH Sets the pen width

Usage: gSETPENWIDTH width%

Sets the pen width in the current drawable to width% pixels.

gSETWIN Changes position/size of the current window

Usage: gSETWIN x%,y%,width%,height%
or gSETWIN x%,y%

Changes position and, optionally, the size of the current window.

An error is raised if the current drawable is a bitmap.

The current position is unaffected.

If you use this command on the default window, you must also use the SCREEN command to ensure that the area for PRINT commands to use is wholly contained within the default window.

gSTYLE Sets the style of text used in subsequent print commands

Usage: gSTYLE style%

Sets the style of text displayed in subsequent gPRINT, gPRINTB, and gPRINTCLIP commands on the current drawable. Values for style%:

KgStyleNormal%	0	normal
KgStyleBold%	1	bold
KgStyleUnder%	2	underline
KgStyleInverse%	4	inverse
KgStyleDoubleHeight%	8	double height
KgStyleMonoFont%	16	monospaced font
KgStyleItalic%	32	italic

You can combine these styles by adding their values, for example, to set bold, underlined, and double height, use:

```
gSTYLE KgStyle-
    Bold%+KgStyleUnder%+KgStyleDoubleHeight%
```

This command does not affect non-graphics commands, like PRINT.

gTMODE Sets the character display mode

Usage: gTMODE mode%

Sets the way characters are displayed by subsequent gPRINT and gPRINTCLIP commands on the current drawable. Values for mode%:

KtModeSet%	0	pixels will be set
KtModeClear%	1	pixels will be cleared
KtModeInvert%	2	pixels will be inverted
KtModeReplace%	3	pixels will be replaced

When you first use graphics text commands on a drawable, each dot in a letter causes a pixel to be set in the drawable, i.e. the default gTMODE is KtModeSet%.

When mode% is Clear or Invert, graphics text commands work in a similar way, but the pixels are cleared or inverted. When mode% is Replace, entire character boxes are drawn on the screen – pixels are set in the letter and cleared in the background box.

This command does not affect other text display commands.

Constants for the modes are supplied in Const.oph.

gTWIDTH Calculates the screen width of a string

Usage: width%=gTWIDTH(text$)

Returns the width of text$ in the current font and style.

See also gPRINT, gPRINTB, gPRINTCLIP, gXPRINT.

gUNLOADFONT Unloads a font

Usage: gULOADFONT fileId%

Unloads a user-defined font that was previously loaded using gLOADFONT. Raises an error if the font has not been loaded.

The built-in fonts are not held in memory and cannot be unloaded.

See also gLOADFONT.

gUPDATE Controls screen updates

Usage: any of

```
gUPDATE ON
gUPDATE OFF
gUPDATE
```

The screen is usually updated whenever you display anything on it. gUPDATE OFF switches off this feature. The screen will then be updated as few times as possible (though note that some keywords will always cause an update). You can still force an update by using the gUPDATE command on its own.

This can result in a considerable speed improvement in some cases. You might, for example, use gUPDATE OFF, then a sequence of graphics commands, followed by gUPDATE. You should certainly use gUPDATE OFF if you are about to write exclusively to bitmaps.

gUPDATE ON returns to normal screen updating.

gUPDATE affects anything that displays on the screen. If you are using a lot of PRINT commands, gUPDATE OFF may make a noticeable difference in speed.

Note that with gUPDATE OFF, the location of errors that occur while the procedure is running may be incorrectly reported. For this reason, gUPDATE OFF is best used in the final stages of program development, and even then you may have to remove it to locate some errors.

gUSE — Sets the current drawable

Usage: gUSE id%

Makes the drawable id% current. Graphics drawing commands will now go to this drawable. gUSE does not bring a drawable to the foreground (see gORDER).

gVISIBLE — Sets the visibility of the current window

Usage: gVISIBLE ON
or gVISIBLE OFF

Makes the current window visible or invisible.

Raises an error if the current drawable is a bitmap.

gWIDTH — Current drawable's width

Usage: width%=gWIDTH

Returns the width of the current drawable.

gX — X position in the current drawable

Usage: x%=gX

Returns the x current position (in from the left) in the current drawable.

gXBORDER — Draws a border

Usage: gXBORDER type%,flags%,w%,h%
or gXBORDER type%,flags%

Draws a border in the current drawable of a specified type, fitting inside a rectangle of the specified size or with the size of the current drawable if no size is specified.

type% should always be the constant value KgXBorderS5Type%. Other border types are implemented for backwards compatibility, but should not be used with Symbian OS.

KgXBorderS5Type%	2	the standard border type

Values for flags% are:

None	$00
Single black	$01
Shallow sunken	$42
Deep sunken	$44
Deep sunken with outline	$54
Shallow raised	$82
Deep raised	$84
Deep raised with outline	$94
Vertical bar	$22
Horizontal bar	$2a

Constants for these flags and types are supplied in Const.oph. The following values of flags% apply to all border types:

0 for normal corners
Adding $100 leaves 1 pixel gap around the border
Adding $200 for more rounded corners
Adding $400 for losing a single pixel
If both $400 and $200 are mistakenly supplied, $200 has priority

See also gBORDER.

gXPRINT Prints a string with precise highlighting or underlining

Usage: gXPRINT string$,flags%

Displays string$ at the current position, with precise highlighting or underlining. The current font and style are still used, even if the style itself is inverse or underlined. If text mode 3 (Replace) is used, both set and cleared pixels in the text are drawn.

Possible values for flags% are:

KgXPrintNormal%	0	normal, as with gPRINT
KgXPrintInverse%	1	inverse
KgXPrintInverseRound%	2	inverse, omitting corner pixels
KgXPrintThinInverse%	3	thin inverse
KgXPrintThinInverseRound%	4	thin inverse, omitting corner pixels
KgXPrintUnderlined%	5	underlined
KgXPrintThinUnderlined%	6	thin underlined

These constants are supplied in Const.oph.

Where lines of text are separated by a single pixel, the thin options maintain the separation between lines.

gXPRINT does not support the display of a list of expressions of various types.

gY — Y position in the current drawable

Usage: y%=gY

Returns the y current position (down from the top) in the current drawable.

HEX$ — Converts an integer to a hex string

Usage: h$=HEX$(x&)

Returns a string containing the hexadecimal (base 16) representation of integer or long integer x&.

For example, HEX$(255) returns the string "FF".

Notes

To enter integer hexadecimal constants (16-bit) put a $ in front of them. For example, $FF is 255 in decimal. (Don't confuse this use of $ with string variable names.)

To enter long integer hexadecimal constants (32-bit) put a & in front of them. For example, &FFFFF is 1048575 in decimal.

Counting in hexadecimal is done like this: 0 1 2 3 4 5 6 7 8 9 A B C D E F 10 ... where A stands for decimal 10, B for decimal 11, C for decimal 12 ... up to F for decimal 15. After F comes 10, which is equivalent to decimal 16. To understand numbers greater than hexadecimal 10, again compare hexadecimals with decimals. In these examples, 102 means 10×10, 103 means $10 \times 10 \times 10$, and so on.

253 in decimal is:

$(2 \times 102) + (5 \times 101) + (3 \times 100) = (2 \times 100) + (5 \times 10) + (3 \times 1) = 200 + 50 + 3$

By analogy, &253 in hexadecimal is:

$(\&2 \times 162) + (\&5 \times 161) + (\&3 \times 160) = (2 \times 256) + (5 \times 16) + (3 \times 1) = 512 + 80 + 3 = 595$ in decimal

Similarly, &A6B in hexadecimal is:

$(\&A \times 162) + (\&6 \times 161) + (\&B \times 160) = (10 \times 256) + (6 \times 16) + (11 \times 1) = 2560 + 96 + 11 = 2667$ in decimal

You may also find this table useful for converting between hex and decimal:

Hex	decimal	–
&1	1	= 16^0
&10	16	= 16^1
&100	256	= 16^2
&1 000	4096	= 16^3

For example, &20F9 is:

(2 × &1000)+(0 × &100)+(15 × &10)+9, which in decimal is: (2 × 4096)+(0 × 256)+(15 × 16)+9=8441.

All hexadecimal constants are integers ($) or long integers (&). So arithmetic operations involving hexadecimal numbers behave in the usual way. For example, &3/&2 returns 1, &3/2.0 returns 1.5, 3/$2 returns 1.

HOUR — Gets the current hour

Usage: h%=HOUR

Returns the number of the current hour from the system clock as an integer between 0 and 23.

IABS — Absolute value of an integer expression

Usage: i&=IABS(x&)

Returns the absolute value, i.e. without any sign, of the integer or long integer expression x&.

For example, IABS(-10) is 10.

See also ABS, which returns the absolute value as a floating point value.

ICON — Sets an application's icon

Usage: ICON mbm$

Gives the name of the bitmap file mbm$, also known as an Symbian OS Picture file, to use as the icon for an OPL application.

If the ICON command is not used inside the APP... ENDA structure, then a default icon is used, but the rest of the information in the APP... ENDA construct is still used to specify the other features of the OPL application.

mbm$ is a multi-bitmap file, which can contain up to three bitmap/mask pairs – the sizes are 24, 32, and 48 squares. These different sizes are used for the different zoom levels in the system screen. The sizes are read from the MBM and the most suitable size is zoomed if the exact sizes required are not provided or if some are missing.

In fact, you can use ICON more than once within the APP...ENDA construct. The translator only insists that all icons are paired with a mask of the same size in the final ICON list. This allows you to use pairs of MBMs containing just one bitmap, as produced by the Sketch application. For example, you could specify them individually:

```
APP ...
ICON "icon24.mbm"
ICON "mask24.mbm"
ICON "icon32.mbm"
ICON "mask32.mbm"
ICON "icon48.mbm"
ICON "mask48.mbm"
ENDA
```

or with pairs in each MBM:

```
APP ...
ICON "iconMask24"
ICON "iconMask32"
ICON "iconMask48"
ENDA
```

or with all the bitmaps as just one MBM, as would normally be the case if prepared on the PC using bmconv and aiftool.

This command can only be used between APP and ENDA.

IF...ENDIF Conditional loop

Usage:

```
IF condition1
  ...
ELSEIF condition2
```

```
  ...
ELSE
  ...
ENDIF
```

Does either
the statements following the IF condition

or
the statements following one of the ELSEIF conditions (there may be as many ELSEIF statements as you like, none at all if you want)

or
the statements following ELSE (or, if there is no ELSE, nothing at all). There may be either one ELSE statement or none.

After the ENDIF statement, the lines following ENDIF carry on as normal.

IF, ELSEIFs, ELSE, and ENDIF must be in that order.

Every IF must be matched with a closing ENDIF.

You can also have an IF...ENDIF structure within another, for example:

```
IF condition1
  ...
ELSE
  ...
  IF condition2
    ....
  ENDIF
  ...
ENDIF
```

condition is an expression returning a logical value, for example a<b. If the expression returns logical true (non-zero) then the statements following are executed. If the expression returns logical false (zero) then those statements are ignored.

INCLUDE Includes a header file

Usage: INCLUDE file$

Includes a file, file$, which may contain CONST definitions, prototypes for OPX procedures, and prototypes for module procedures. The included file may not include module procedures themselves. Procedure and

OPX procedure prototypes allow the translator to check parameters and coerce numeric parameters (that are not passed by reference) to the required type.

Including a file is logically identical to replacing the INCLUDE statement with the file's contents.

The filename of the header may or may not include a path. If it does include a path, then OPL will only scan the specified folder for the file. However, the default path for INLCUDE is \System\Opl\, so when INCLUDE is called without specifying a path, OPL looks for the file firstly in the current folder and then in \System\Opl\ in all drives from Y: to A: and then in Z:, excluding any remote drives.

See CONST, EXTERNAL.

INPUT Reads a value from the keyboard

Usage: INPUT variable
or INPUT log.field

Waits for a value to be entered at the keyboard, and then assigns the value entered to a variable or data file field.

You can edit the value as you type it in. All the usual editing keys are available: the arrow keys move along the line, Esc clears the line, and so on.

If inappropriate input is entered, for example a string when the input was to be assigned to an integer variable, a ? is displayed and you can try again. However, if you used TRAP INPUT, control passes on to the next line of the procedure, with the appropriate error condition being set and the value of the variable remaining unchanged.

INPUT is usually used in conjunction with a PRINT statement:

```
PROC exch:
  LOCAL pds,rate
  DO
    PRINT "Pounds Sterling?",
    INPUT pds
    PRINT "Rate (DM)?",
    INPUT rate
```

```
    PRINT "=",pds*rate,"DM"
    GET
  UNTIL 0
ENDP
```

Note the commas at the end of the PRINT statements, used so that the cursor waiting for input appears on the same line as the messages.

TRAP INPUT

If a bad value is entered (for example "abc" for a%) in response to a TRAP INPUT, the ? is not displayed, but the ERR function can be called to return the value of the error that has occurred. If the Esc key is pressed while no text is on the input line, the 'Escape key pressed' error (number −114) will be returned by ERR (provided that the INPUT has been trapped). You can use this feature to enable someone to press the Esc key to escape from inputting a value.

See also EDIT. This works like INPUT, except that it displays a string to be edited and then assigned to a variable or field. It can only be used with strings.

INSERT — Inserts a blank record into a database

Usage: INSERT

Inserts a new, blank record into the current view of a database. The fields can then be assigned to before using PUT or CANCEL.

INT — Gets the integer part of a floating point value (as an integer)

Usage: i&=INT(x)

Returns the integer (in other words the whole number) part of the floating point expression x. The number is returned as a long integer.

Positive numbers are rounded down, and negative numbers are rounded up. For example, INT(-5.9) returns -5 and INT(2.9) returns 2. If you want to round a number to the nearest integer, add 0.5 to it (or subtract 0.5 if it is negative) before you use INT.

See also INTF.

INTF — Gets the integer part of a floating point value (as a floating point number)

Usage: i=INTF(x)

Used in the same way as the INT function, but the value returned is a floating point number. For example, INTF(1234567890123.4) returns 1234567890123.0.

You may also need this when an integer calculation may exceed integer range.

See also INT.

INTRANS — True if the current view is in a transaction

Usage: i&=INTRANS

Finds out whether the current view is in a transaction. Returns −1 if it is in a transaction or 0 if it is not.

See also BEGINTRANS.

IOA — Asynchronous I/O request

Usage: r%=IOA(h%,f%,var status%,var a1,var a2)

This has the same form as IOC, but it returns an error value if the request is not completed successfully. IOC should be used in preference to IOA.

IOC — I/O request with guaranteed completion

Usage: IOC(h%,f%,var status%,var a1,var a2)

Make an I/O request with guaranteed completion. The device driver opened with handle h% performs the asynchronous I/O function f% with two further arguments, a1 and a2. The argument status% is set by the device driver. If an error occurs while making a request, status% is set to an appropriate value, but IOC always returns zero, not an error value. An IOWAIT or IOWAITSTAT must be performed for each IOC. IOC should be used in preference to IOA.

IOCANCEL — Cancels an outstanding IO request

Usage: r%=IOCANCEL(h%)

Cancels any outstanding asynchronous I/O request (IOC or IOA). Note, however, that the request will still complete, so the signal must be consumed using IOWAITSTAT.

IOCLOSE Closes a file

Usage: r%=IOCLOSE(h%)

Closes a file with the handle h%.

See also I/O functions and commands.

IOOPEN Creates or opens a file

Usage: r%=IOOPEN(var h%,name$,mode%)

Creates or opens a file called name$. Defines h% for use by other I/O functions. mode% specifies how to open the file. For unique file creation, use IOOPEN(var h%,addr%,mode%).

See also I/O functions and commands.

IOREAD Reads from a file

Usage: r%=IOREAD(h%,addr&,maxLen%)

Reads from the file with the handle h%. address% is the address of a buffer large enough to hold a maximum of maxLen% bytes. The value returned to r% is the actual number of bytes read or, if negative, is an error value.

IOSEEK Seeks to a position in a file opened for random access

Usage: r%=IOSEEK(h%,mode%,var off&)

Seeks to a position in a file that has been opened for random access. mode% specifies how the offset argument off& is to be used. Values for mode% may be found in the I/O functions and commands. off& may be positive to move forwards or negative to move backwards. IOSEEK sets the variable off& to the absolute position set.

Note the following example when you use #:

```
ret%=IOSEEK(h%,mode%,#ptrOff&)
```

passing the long integer ptrOff&.

IOSIGNAL Signals completion of asynchronous I/O function

Usage: IOSIGNAL

Signals an asynchronous I/O function's completion.

IOW — Synchronous I/O request

Usage: r%=IOW(h%,func%,var a1,var a2)

The device driver opened with handle h% performs the synchronous I/O function func% with the two further arguments.

IOWAIT — Waits for an asynchronous I/O function to complete

Usage: IOWAIT

Waits for an asynchronous I/O function to signal completion.

IOWAITSTAT — Waits for an IOC or IOA to complete

Usage: IOWAITSTAT var stat%

Waits for an asynchronous function, called with IOC or IOA, to complete.

IOWAITSTAT32 — Waits for asynchronous OPX procedure 32-bit status word to complete

Usage: IOWAITSTAT32 var stat&

Takes a 32-bit status word. IOWAITSTAT32 should be called only when you need to wait for completion of a request made using a 32-bit status word when calling an asynchronous OPX procedure.

Note: The initial value of a 32-bit status word while it is still pending (i.e. waiting to complete) is &80000001 (KStatusPending32& in Const.oph. For a 16-bit status word the 'pending value' is −46 (KErrFilePending%).

IOWRITE — Writes bytes in a buffer to a file

Usage: r%=IOWRITE(h%,addr&,length%)

Writes length% bytes in a buffer at address% to the file with the handle h%.

IOYIELD — Ensures asynchronous functions have a chance to run

Usage: IOYIELD

Ensures that any asynchronous handler set up with IOC or IOA is given a chance to run. IOYIELD must always be called before polling status words, i.e. before reading

a 16-bit status word if IOWAIT or IOWAITSTAT have not been used first.

KEY — Gets the last key pressed as a character code

Usage: k%=KEY

Returns the character code of the key last pressed, if there has been a key press since the last use of the keyboard by INPUT, EDIT, GET, GET$, KEY, KEY$, MENU, and DIALOG.

If no key has been pressed, zero is returned.

See Character codes for a list of special key codes. You can use KMOD to check whether modifier keys (Shift, Ctrl, Fn, and Caps Lock) were used.

This command does not wait for a key to be pressed, unlike GET.

KEY$ — Gets the last key pressed as a string

Usage: k$=KEY$

Returns the last key pressed as a string, if there has been a key press since the last use of the keyboard by INPUT, EDIT, GET, GET$, KEY, KEY$, MENU, and DIALOG.

If no key has been pressed, a null string ("") is returned.

See Character codes for a list of special key codes. You can use KMOD to check whether modifier keys (Shift, Ctrl, Fn, and Caps Lock) were used.

This command does not wait for a key to be pressed, unlike GET$.

KEYA — Reads the keyboard asynchronously

Usage: err%=KEYA(var stat%,var key%(1))

This is an asynchronous keyboard read function.

Cancel with KEYC.

KEYC — Cancels a KEYA

Usage: err%=KEYC(var stat%)

Cancels the previously called KEYA function with status stat%. Note that KEYC consumes the signal (unlike IOCANCEL), so IOWAITSTAT should not be used after KEYC.

KILLMARK Removes a bookmark

Usage: KILLMARK b%

Removes the bookmark b%, which has previously been returned by BOOKMARK, from the current view of a database.

See BOOKMARK, GOTOMARK.

KMOD Gets the state of the modifier keys

Usage: k%=KMOD

Returns a code representing the state of the modifier keys (whether they were pressed or not) at the time of the last keyboard access, such as a KEY function. The modifiers have these codes:

KKmodShift%	2	Shift down
KKmodControl%	4	Ctrl down
KKmodCaps%	16	Caps lock on
KKmodFn%	32	Fn down

These constants are supplied in Const.oph.

If there was no modifier, the function returns 0. If a combination of modifiers was pressed, the sum of their codes is returned – for example, 20 is returned if Ctrl (4) was held down and Caps lock (16) was on.

Always use immediately after a KEY/KEY$/GET/GET$ statement.

The value returned by KMOD has one binary bit set for each modifier, as shown above. By using the logical operator AND on the value returned by KMOD you can check which of the bits are set, in order to see which modifier keys were held down. For more details on AND, see Operators and logical expressions.

Example:

```
PROC modifier:
  LOCAL k%,mod%
  PRINT "Press a key" :k%=GET
  CLS :mod%=KMOD
  PRINT "Key code",k%,"with"
  IF mod%=0
    PRINT "no modifier"
  ENDIF
```

```
  IF mod% AND KModShift%
    PRINT "Shift down"
  ENDIF
  IF mod% AND KModControl%
    PRINT "Control down"
  ENDIF
  IF mod% AND KModCaps%
    PRINT "Caps Lock on"
  ENDIF
  IF mod% AND KModFn%
    PRINT "Fn down"
  ENDIF
ENDP
```

LAST — Positions to the last record

Usage: LAST

Positions to the last record in a view.

LCLOSE — Closes the device opened with LOPEN

Usage: LCLOSE

Closes the device opened with LOPEN. The device is also closed automatically when a program ends.

LEFT$ — Gets the leftmost characters of a string

Usage: b$=LEFT$(a$,x%)

Returns the leftmost x% characters from the string a$.

For example, if n$ has the value Charles, then b$=LEFT$(n$,3) assigns Cha to b$.

LEN — Length of a string

Usage: a%=LEN(a$)

Returns the number of characters in a$.

E.g. if a$ has the value 34 Kopechnie Drive then LEN(a$) returns 18.

You might use this function to check that a data file string field is not empty before displaying:

```
IF LEN(A.client$)
  PRINT A.client$
ENDIF
```

ENALLOC Gets the length of a previously allocated cell

Usage: len&=LENALLOC(pcell&)

Returns the length of the previously allocated cell at pcell&. An error will be raised if the cell address argument is not in the range known by the heap.

See also SETFLAGS if you require a 64K memory limit to be enforced. If the flag is set to restrict the limit, len& is guaranteed to fit into an integer.

ARM

Cells are allocated lengths that are the smallest multiple of four greater than the size requested because the ARM processor requires a 4-byte word alignment for its memory allocation.

LN Natural logarithm

Usage: a=LN(x)

Returns the natural (base e) logarithm of x.

Use LOG to return the base 10 log of a number.

LOADM Loads a translated OPL module

Usage: LOADM module$

Loads a translated OPL module so that procedures in that module can be called. Until a module is loaded with LOADM, calls to procedures in that module will give an error.

module$ is a string containing the name of the module. Specify the full filename only where necessary.

Example: LOADM "MODUL2"

Up to 8 modules can be in memory at any one time, including the top-level module; if you try to LOADM a ninth module, you get an error. Use UNLOADM to remove a module from memory so that you can load a different one.

By default, LOADM always uses the folder of the top-level module. It is not affected by the SETPATH command.

LOC Locates a substring within a string

Usage: a%=LOC(a$,b$)

Returns an integer showing the position in a$ where b$ occurs, or zero if b$ doesn't occur in a$. The search matches upper and lowercase.

Example: LOC("STANDING","AND") would return the value 3 because the substring AND starts at the third character of the string STANDING.

LOCAL Declares procedure-local variables

Usage: LOCAL variables

Used to declare variables that can be referenced only in the current procedure. Other procedures may use the same variable names to create new variables. Use GLOBAL to declare variables common to all called procedures.

The variables may be of four types, depending on the symbol they end with:

Variable names not ending with $, %, &, or () are floating point variables, for example price, x
Those ending with a % are integer variables, for example x%, sales92%
Those ending with an & are long integer variables, for example x&, sales92&
Those ending with a $ are string variables. String variable names must be followed by the maximum length of the string in brackets, for example names$(12), a$(3)

Array variables have a number immediately following them in brackets, which specifies the number of elements in the array. Array variables may be of any type, for example: x(6),y%(5),f$(5,12),z&(3).

When declaring string arrays, you must give two numbers in the brackets. The first declares the number of elements, the second declares their maximum length. For example, surname$(5,8) declares five elements, each up to eight characters long.

Variable names may be any combination of up to 32 numbers, alphabetic letters, and the underscore character. They must start with a letter or an underscore. The length includes the %, &, or $ sign, but not the () in string and array variables.

More than one GLOBAL or LOCAL statement may be used, but they must be on separate lines, immediately after the procedure name.

See also GLOBAL, CONST.

LOCK — Blocks system requests to change files or quit

Usage: LOCK ON
or LOCK OFF

Marks an application as locked or unlocked. When an app is locked with LOCK ON, the System screen will not send it events to change files or quit.

If, for example, you move to the task list or the document name in the System screen, try to stop that running app by using the 'Close file' button or Ctrl+E, a message will appear, indicating that the app cannot close down at that moment.

You should use LOCK ON if your application uses a command, such as EDIT, MENU, or DIALOG, which accesses the keyboard. You might also use it when the app is about to go busy for a considerable length of time, or at any other point where a clean exit is not possible. Do not forget to use LOCK OFF as soon as possible afterwards.

An application is initially unlocked.

LOG — Logarithm

Usage: a=LOG(x)

Returns the base 10 logarithm of x.

Use LN to find the base e (natural) log.

LOPEN — Opens a device for printing

Usage: LOPEN device$

Opens the device to which LPRINTs are to be sent.

No LPRINTs can be sent until a device has been opened with LOPEN.

You can open any of these devices:

The serial port, with LOPEN "TTY:A"

A file on the Symbian OS device. Any existing file of the name given will be overwritten when you print to it

Only one device may be open at any one time. Use LCLOSE to close the device. LOPENned devices also close automatically when a program finishes running.

LOWER$ — Converts a string to lowercase

Usage: b$=LOWER$(a$)

Converts any uppercase characters in the string a$ to lowercase and returns the completely lowercase string.

E.g. if a$="CLARKE", LOWER$(a$) returns the string "clarke".

Use UPPER$ to convert a string to uppercase.

LPRINT — Prints a list to the device opened using LOPEN

Usage: LPRINT list of expressions

Prints a list of items, in the same way as PRINT, except that the data is sent to the device most recently opened with LOPEN.

The expressions may be quoted strings, variables, or the evaluated results of expressions. The punctuation of the LPRINT statement (commas, semicolons, and new lines) determines the layout of the printed text, in the same way as PRINT statements.

If no device has been opened with LOPEN you will get an error.

See PRINT for displaying to the screen.

See LOPEN for opening a device for LPRINT.

MAX — Maximum value

Usage: m=MAX(list)
or m=MAX(array(),element)

Returns the greatest of a list of numeric items. The list can be either:

A list of variables, values, and expressions, separated by commas

or

The elements of a floating point array.

When operating on an array, the first argument must be the array name followed by (). The second argument, separated from the first by a comma, is the number of array elements you wish to operate on. For example, m=MAX(arr(),3) would return the value of the largest of elements arr(1), arr(2), and arr(3).

mCARD — Defines a menu card

Usage: mCARD title$,n1$,k1%
or mCARD title$,n1$,k1%,n2$,k2% etc.

Defines a menu. When you have defined all of the menus, use MENU to display them.

title$ is the name of the menu. From one to eight items on the menu may be defined, each specified by two arguments. The first is the item name, and the second the keycode for a shortcut key. This specifies a key which, when pressed together with Ctrl, will select the option. If the keycode is for an uppercase key, the shortcut key will be Shift+Ctrl.

The options can be divided into logical groups by displaying a separating line under the final option in a group. To do this, pass the negative value corresponding to the shortcut key keycode for the final option in the group. For example, – %A specifies shortcut key Shift+Ctrl+A and displays a separating line under the associated option in the menu.

Menu items without shortcuts can be specified using shortcut values between 1 and 32. For these items the value specified is still returned if the item is selected. Any item with a shortcut of zero will NOT be added to the menu.

Other properties can be specified by adding one or more of the following flags to the shortcut keycode:

KMenuDimmed%	$1000	menu item is dimmed
KMenuSymbolOn%	$2000	checkbox option button symbol on
KMenuSymbolIndeterminate%	$4000	checkbox or option button symbol indeterminate
KMenuCheckBox%	$0800	item has a checkbox
KMenuOptionStart%	$0900	item starts an option button list
KMenuOptionMiddle%	$0A00	in the middle of an option button list
KMenuOptionEnd%	$0B00	ends an option button list

These constants are supplied in Const.oph.

The start, middle, and end option buttons are for specifying a group of related items that can be selected exclusively (i.e. if one item is selected then the others are deselected). The number of middle option buttons is variable. A single menu card can have more than one set of option buttons and checkboxes, but option buttons in a set should be kept together. For speed,

OPL does not check the consistency of these items' specification.

If a separating line is required when any of these effects have been added, you must be sure to negate the whole value, not just the shortcut key keycode. For example:

```
mCARD "Options","View1",%A OR $2900,"View2",-
    (%B OR $B00), "Another option",%C
```

Here, the second shortcut key keycode and its flag value is correctly negated to display a separating line.

A 'Too wide' error is raised if the menu title length is greater than or equal to 40. Shortcut values must be alphabetic character codes or numbers between the values of 1 and 32. Any other values will raise an 'Invalid arguments' error.

If any menu item fails to be added successfully, a menu is discarded. It is therefore incorrect to ignore mCARD errors by having an ONERR label around an mCARD call. If you do, the menu is discarded and a 'Structure fault' will be raised on using mCARD without first using mINIT again. See MENU for an example of this.

See also mCARDX.

mCARDX Defines a menu card with graphic

Usage: mCARDX BitmapID&, BitmapMaskID&, Item1$, Item1key%, Item2$, Item2Key%

The mCARDX keyword is essentially the same as mCARD but with the difference that instead of supplying a string caption for the menu card, you supply a bitmap and bitmap mask instead.

This above example will add a pane to the menu with the images in BitmapID& and BitmapMaskID& as its caption. Note that for Series 80 (where this command is normally used), the recommended size for these bitmaps is 25×20 pixels in 256 colors (i.e. specify the/c8 flag when using the BMCONV tool provided on all Symbian OS v6.0 SDKs).

See also mCARD.

mCASC Defines a menu cascade

Usage: mCASC title$,item1$,hotkey1%,item2$,
hotkey2%

Creates a cascade for a menu, on which less important menu items can be displayed. The cascade must be defined before use in a menu card. For example, a 'Bitmap' cascade under the File menu of a possible OPL drawing application could be defined like this:

```
mCASC "Bitmap","Load",%L,"Merge",%M
mCARD "File","New",%n,"Open",%o,"Save",%s,
    "Bitmap>",16,"Exit",%e
```

The trailing > character specifies that a previously defined cascade item is to be used in the menu at this point: it is not displayed in the menu item. A cascade has a filled arrow head displayed along side it in the menu. The cascade title in mCASC is also used only for identification purposes and is not displayed in the cascade itself. This title needs to be identical to the menu item text apart from the >. For efficiency, OPL doesn't check that a defined cascade has been used in a menu and an unused cascade will simply be ignored. To display a > in a cascaded menu item, you can use ≫.

Shortcut keys used in cascades may be added to the appropriate constant values as for mCARD to enable checkboxes, option buttons, and dimming of cascade items.

As is typical for cascade titles, a shortcut value of 16 is used in the example above. This prevents the display or specification of any shortcut key. However, it is possible to define a shortcut key for a cascade title if required, for example to cycle through the options available in a cascade.

If more menu items are displayed in a menu card or cascade than are displayable on the screen, the menu will be drawn with a scroll bar. It is recommended that programmers do not define menus that scroll to keep the pen actions the user has to make to use the program to a minimum: see Defining the menus.

See also mCARD, mCARDX, MENU, mINIT.

MEAN Calculates a mean value

Usage: m=MEAN(list)
or m=MEAN(array(),element)

Returns the arithmetic mean (average) of a list of numeric items. The list can be either:

A list of variables, values, and expressions, separated by commas

or

The elements of a floating point array.

When operating on an array, the first argument must be the array name followed by (). The second argument, separated from the first by a comma, is the number of array elements you wish to operate on. For example, m=MEAN(arr(),3) would return the average of elements arr(1), arr(2), and arr(3).

This example displays 15.0:

```
a(1)=10
a(2)=15
a(3)=20
PRINT MEAN(a(),3)
```

MENU Displays a menu

Usage: val%=MENU
or val%=MENU(var init%)

Displays the menus defined by mINIT, mCARD, and mCASC, and waits for you to select an item. Returns the shortcut key keycode of the item selected, as defined in mCARD, in lowercase.

If you cancel the menu by pressing Esc, MENU returns 0.

If the name of a variable is passed, it sets the initial menu pane and item to be highlighted. init% should be 256*(menu%)+item%; for both menu% and item%, 0 specifies the first, 1 the second, and so on. If init% is 517 (=256*2+5), for example, this specifies the sixth item on the third menu.

If init% was passed, MENU writes back to init% the value for the item that was last highlighted on the menu. You can then use this value when calling the menu again.

It is necessary to use MENU(init%), passing back the same variable each time the menu is opened if you wish the menu to reopen with the highlight set on the last selected item.

It is incorrect to ignore mCARD and mCASC errors by having an ONERR label around an mCARD or mCASC call. If you do, the menu is discarded and a 'Structure fault' will be raised on using mCARD, mCASC, or MENU without first using mINIT again.

The following bad code will not display the menu:

```
REM ** example of bad code **
  mINIT
  ONERR errIgnore1
  mCARD "Xxx","ItemA",0   REM bad shortcut
errIgnore1::
  ONERR errIgnore2
  mCARD "Yyy",""
  REM 'Structure fault' error (mINIT discarded)
errIgnore2::
  ONERR OFF
  MENU            REM 'Structure fault' again
```

See also mCARD, mCASC, mINIT.

MID$ Gets the middle part of a string

Usage: m$=MID$(a$,x%,y%)

Returns a string comprising y% characters of a$, starting at the character at position x%.

E.g. if name$="McConnell" then MID$(name$,3,4) would return the string Conn.

MIN Minimum value

Usage: m=MIN(list)
or m=MIN(array(),element)

Returns the smallest of a list of numeric items. The list can be either:

A list of variables, values, and expressions, separated by commas

or

The elements of a floating point array.

When operating on an array, the first argument must be the array name followed by (). The second argument, separated from the first by a comma, is the number of array elements you wish to operate on. For example,

m=MIN(arr(),3) would return the minimum of elements arr(1), arr(2), and arr(3).

mINIT — Initializes menus

Usage: mINIT

Prepares for definition of menus, cancelling any existing menus. Use mCARD and mCASC to define each menu, then MENU to display them.

It is incorrect to ignore mCARD or mCASC errors by having an ONERR label around an mCARD or mCASC call. If you do, the menu is discarded and a 'Structure fault' will be raised if there is an occurrence of mCARD, mCASC, or MENU without first using mINIT again. See MENU for an example of this.

See also mCARD, mCARDX, mCASC and Menus.

MIME — Associates an OPL application with a MIME type

Usage: MIME pri%, dtype$

Associates an OPL application with the Internet MIME content type dtype$, using priority pri%. This command can only be used in the scope of an APP...ENDA construct. The priority value specifies how proficient the application is at handling the named data type. dtype% is the name of the data type this app is declaring that it can handle, e.g. text/html, image/png, or text/plain.

Declaring a MIME association indicates to the system that the application allows the user to view or manipulate files and data of the named type.

pri% can take any of the following values:

KDataTypePriorityUserSpecified%	KMaxInt%	reserved for future use
KDataTypePriorityHigh%	10000	this app is superbly capable of handling his data type
KDataTypePriorityNormal%	0	typical priority. App is proficient at handling this data type
KDataTypePriorityLow%	−10000	app is merely capable of handling this document type

KDataTypePriorityLastResort%	−20000	app should only handle this data type if there are no other apps available that can use it
KDataTypePriorityNotSupported%	KMinInt%	–

These constants are supplied in Const.oph. Note that KDataTypePriorityUserSpecified% is reserved for future use.

The .aif file (application information file) for the app is used to store this information once the app has been translated. Only one MIME association is allowed per application, so only one MIME statement can be made in an OPL application's APP...ENDA declaration.

There are recognizers in the ROM for the following MIME types:

text/plain
text/html
image/jpeg
image/gif
text/X-vCard
text/X-vCalendar

MINUTE — Gets the current minute

Usage: m%=MINUTE

Returns the current minute number from the system clock (0 to 59).

E.g. at 8.54 a.m. MINUTE returns 54.

MKDIR — Creates a new folder

Usage: MKDIR name$

Creates a new folder/directory. For example, MKDIR "C:\MINE\TEMP" creates a C:\MINE\TEMP folder, also creating C:\MINE if it is not already there.

MODIFY — Changes a record without moving it

Usage: MODIFY

Allows the current record of a view to be modified without moving the record. The fields can then be assigned to before using PUT or CANCEL.

MONTH Gets the current month

Usage: m%=MONTH

Returns the current month from the system clock as an integer between 1 and 12.

E.g. on 12th March 1992 m%=MONTH returns 3 (KMarch%) to m%.

MONTH$ Converts a numeric month to a string

Usage: m$=MONTH$(x%)

Converts x%, a number from 1 to 12, to the month name, expressed as a three-letter mixed case string.

E.g. MONTH$(KJanuary%) returns the string Jan.

See MONTH for the month number constants.

mPOPUP Presents a popup menu

Usage: mPOPUP(x%,y%,posType%,item1$,key1%,
item2$,key2%,...)

Presents a popup menu. mPOPUP returns the value of the key press used to exit the popup menu, this being 0 if Esc is pressed.

Note that mPOPUP defines and presents the menu itself, and should not and need not be called from inside the mINIT...MENU structure.

posType% is the position type controlling which corner of the popup menu x%,y% specifies and can take the values:

KMPopupPosTopLeft%	0	top left
KMPopupPosTopRight%	1	top right
KMPopupPosBottomLeft%	2	bottom left
KMPopupPosBottomRight%	3	bottom right

These constants are supplied in Const.oph.

item$ and key% can take the same values as for mCARD, with key% taking the same constant values to specify checkboxes, option buttons, and dimmed items. However, cascades in popup menus are not supported.

For example:

```
mPOPUP (0,0,0,"Continue",%c,"Exit",%e)
```

specifies a popup menu with 0,0 as its top left-hand corner, with the items 'Continue' and 'Exit', with the shortcut keys Ctrl+C and Ctrl+E, respectively.

See also mCARD.

NEXT — Positions to the next record

Usage: NEXT

Positions to the next record in the current data file.

If NEXT is used after the end of a file has been reached, no error is reported but the current record is null and the EOF function returns true.

NUM$ — Converts a floating point number to a string

Usage: n$=NUM$(x,y%)

Returns a string representation of the integer part of the floating point number x, rounded to the nearest whole number. The string will be up to y% characters wide.

If y% is negative then the string is right-justified, for example NUM$(1.9,−3) returns "2" where there are two spaces to the left of the 2.

If y% is positive no spaces are added: e.g. NUM$(−3.7,3) returns "−4".

If the string returned to n$ will not fit in the width y%, then the string will just be asterisks; for example NUM$(256.99,2) returns "**".

See also FIX$, GEN$, SCI$.

ONERR — Establishes an error handler

Usage:

```
ONERR label::
...
ONERR OFF
```

or just:

```
ONERR label
...
ONERR OFF
```

ONERR label:: establishes an error handler in a procedure. When an error is raised, the program jumps to the label:: instead of the program stopping and an error message being displayed.

The label may be up to 32 characters long starting with a letter or an underscore. It ends with a double colon (::), although you don't need to use this in the ONERR statement.

ONERR OFF disables the ONERR command, so that any errors occurring after the ONERR OFF statement no longer jump to the label.

It is advisable to use the command ONERR OFF immediately after the label:: which starts the error handling code.

OPEN Opens an existing table in a database

Usage: OPEN query$,log,f1,f2, ...

Opens an existing table (or a 'view' of a table) from an existing database, giving it the logical view name log and handles for the fields f1, f2. log can be any letter in the range A to Z.

query$ specifies the database file, the required table and fields to be selected.

For example:

```
OPEN "clients SELECT name, tel FROM
    phone",D,n$,t$
```

The database name here is clients and the table name is phone. The field names are enclosed by the keywords SELECT and FROM and their types should correspond with the list of handles (i.e. n$ indicates that the name field is a string).

Replacing the list of field names with * selects all the fields from the table.

query$ is also used to specify an ordered view and if a suitable index has been created, then it will be used.

```
OPEN "people SELECT name,number FROM
    phoneBook ORDER BY name
   ASC, number DESC",G,n$,num%
```

would open a view with name fields in ascending alphabetical order and if any names were the same then the number field would be used to order these records in descending numerical order.

If the specification of the database includes embedded spaces, for example in the name of the folder, the name must be enclosed in quotes, so for example the following correctly fails:

```
OPEN "c:\folder with spaces\file with
    spaces",a,name$
```

whereas the following works:

```
OPEN """c:\folder with spaces\file with
    spaces""",a,name$
```

See also CREATE, USE, OPENR.

OPENR — Opens a table as read-only in an existing database

Usage: OPEN query$,log,f1,f2, ...

This command works exactly like OPEN except that the opened file is read-only. In other words, you cannot APPEND, UPDATE, or PUT the records it contains.

This means that you can run two separate programs at the same time, both sharing the same file.

PARSE$ — Parses a filename

Usage: p$=PARSE$(f$,rel$,var off%())

Returns a full file specification from the filename f$, filling in any missing information from rel$.

The offsets to the filename components in the returned string is returned in off%(), which must be declared with at least 6 integers. Index values for off%() are:

KParseAOffFSys%	1	filing system name offset
KParseAOffDev%	2	device name offset
KParseAOffPath%	3	path offset
KParseAOffFilename%	4	filename offset
KParseAOffExt%	5	file extension offset
KParseAOffWild%	6	flags for wildcards in returned string. See below

The flag values in offset%(KParseAOffWild%) are:

KParseWildNone%	0	no wildcards
KParseWildFilename$	1	wildcard in filename
KParseWildExt$	2	wildcard in file extension
KParseWildBoth$	3	wildcard in both

These constants are supplied in Const.oph.

If rel$ is not itself a complete file specification, the current filing system, device, and/or path are used as necessary to fill in the missing parts.

f$ and rel$ should be separate strings.

```
p$=PARSE$("NEW","C:\Documents\*.MBM",x%())
```

sets p$ to C:\Documents\NEW.MBM and x%() to (1,1,3,14,17,0).

PAUSE — Waits for a length of time

Usage: PAUSE x%

Pauses the program for a certain time, depending on the value of x%:

0	waits for a key to be pressed
+ve	pauses for x% twentieths of a second
-ve	pauses for x% twentieths of a second or until a key is pressed

So PAUSE 100 would make the program pause for 100/20 = 5 seconds, and PAUSE −100 would make the program pause for 5 seconds or until a key is pressed.

If x% is less than or equal to 0, a GET, GET$, KEY, or KEY$ will return the key press that terminated the pause. If you are not interested in this key press, but in the one that follows it, clear the buffer after the PAUSE with a single KEY function: PAUSE −10 :KEY

You should be especially careful about this if x% is negative, since then you cannot tell whether the pause was terminated by a key press or by the time running out.

PAUSE should not be used in conjunction with GETEVENT or GETEVENT32 because events are discarded by PAUSE.

PEEKB Reads a short integer from a byte of memory

```
p%=PEEKB(x&)
```

The PEEK functions find the values stored in specific bytes. PEEKB returns the integer value of the byte at address x&.

PEEKW Reads an integer from memory

```
p%=PEEKW(x&)
```

Returns the integer at address x&.

PEEKL Reads a long integer from memory

```
p&=PEEKL(x&)
```

Returns the long integer value at address x&.

PEEKF Reads a floating point value from memory

```
p=PEEKF(x&)
```

Returns the floating point value at address x&.

PEEK$ Reads a string from memory

```
p$=PEEK$(x&)
```

Returns the string at address x&.

PI Returns the value of PI

Usage: p=PI

Returns the value of (3.14...).

POINTERFILTER Sets the pointer event mask

Usage: POINTERFILTER filter%,mask%

Filters pointer events in the current window out or back in. Add the following flags together to achieve the

desired filter% and mask%:

	$0	none
KPointerFilterEnterExit%	$1	enter/exit
KPointerFilterMove%	$2	move
KPointerFilterDrag%	$4	drag

These constants are supplied in Const.oph.

The bits set in filter% specify the settings to be used, 1 to filter out the event and 0 to remove the filter. Only those bits set in mask% will be used for filtering. This allows the current setting of a particular bit to be left unchanged if that bit is zero in the mask (i.e. mask% dictates what to change and filter% specifies the setting to which it should be changed). For example:

```
mask% = KPointerFilterEnterExit% +
    KPointerFilterDrag%
REM  allows enter/exit and drag settings to be
    changed
POINTERFILTER KPointerFilterEnterExit%, mask%
REM filters out enter/exit, but not dragging
...
POINTERFILTER KPointerFilterDrag%, mask%
REM filters out drag and reinstates enter/exit
```

Initially the events are not filtered out.

See also GETEVENT32, GETEVENTA32.

POKEB Stores a short integer in a byte of memory

```
POKEB x&,y%
```

The POKE commands store values in specific bytes. POKEB stores the integer value y% (less than 256) in the single byte at address x&.

POKEW Stores an integer in memory

```
POKEW x&,y%
```

Stores the integer y% across two consecutive bytes, with the least significant byte in the lower address, that is x&.

POKEL Stores a long integer in memory

```
POKEL x&,y&
```

Stores the long integer y& in bytes starting at address x&.

POKEF — Stores a floating point value in memory

```
POKEF x&,y
```

Stores the floating point value y in bytes starting at address x&.

POKE$ — Stores a string in memory

```
POKE$ x&,y$
```

Stores the string y$ in bytes starting at address x&.

Use ADDR to find out the address of your declared variables.

POS — Gets the position in the current view

Usage: p%=POS

Returns the number of the current record in the current view. POS (and POSITION) exist mainly for compatibility with older versions of OPL and you are advised to use bookmarks instead.

A file has no limit on the number of records. However, integers can only be in the range −32768 to +32767. Record numbers above 32767 are therefore returned like this:

record value	returned by POS
32767	32767
32768	32768
32769	32767
32770	32766
...	...
65534	2

To display record numbers, you can use this check:

```
IF POS<0
  PRINT 65536+POS
ELSE
```

```
  PRINT POS
ENDIF
```

Note: The number of the current record may be greater than or equal to 65535, and hence values may need to be truncated to fit into p%, giving inaccurate results. You are particularly advised to use bookmarks when dealing with a large number of records. Note, however, that the value returned by POS can become inaccurate if used in conjunction with bookmarks and multiple views on a table. Accuracy can be restored by using FIRST or LAST on the current view.

See BOOKMARK, GOTOMARK, KILLMARK.

POSITION — Sets the position in the current view

Usage: POSITION x%

Makes record number x% the current record in the current view. By using bookmarks and editing the same table via different views, positional accuracy can be lost and POSITION x% could access the wrong record. Accuracy can be restored by using FIRST or LAST on the current view.

POSITION (and POS) exist mainly for compatibility with older versions of OPL and you are advised to use bookmarks instead.

See BOOKMARK, GOTOMARK, KILLMARK.

PRINT — Displays a list of expressions

Usage: PRINT list of expressions

Displays a list of expressions on the screen. The list can be punctuated in one of these ways:

If items to be displayed are separated by commas, there is a space between them when displayed.

If they are separated by semicolons, there are no spaces.

Each PRINT statement starts a new line, unless the preceding PRINT ended with a semicolon or comma.

There can be as many items as you like in this list. A single PRINT on its own just moves to the next line.

Example: On 1st January 1997

Code	display
PRINT "TODAY is", :PRINT DAY;".";MONTH;".";YEAR	TODAY is 1.1.1997
PRINT 1	1
PRINT "Hello"	Hello
PRINT "Number",1	Number 1

See also LPRINT, gUPDATE, gPRINT, gPRINTB, gPRINT-CLIP, gXPRINT.

PUT — Writes changes into a database

Usage: PUT

Marks the end of a database's INSERT or MODIFY phase and makes the changes permanent.

See INSERT, MODIFY, CANCEL.

RAD — Converts from degrees to radians

Usage: r=RAD(x)

Converts x from degrees to radians.

All the trigonometric functions assume angles are specified in radians, but it may be easier for you to enter angles in degrees and then convert with RAD.

Example:

```
PROC xcosine:
  LOCAL angle
  PRINT "Angle (degrees)?:";
  INPUT angle
  PRINT "COS of",angle,"is",
  angle=RAD(angle)
  PRINT COS(angle)
  GET
ENDP
```

(The formula used is (PI*x)/180.)

To convert from radians to degrees use DEG.

RAISE Raises an error

Usage: RAISE x%

Raises an error.

The error raised is error number x%. This may be one of the errors listed in OPL error values, or a new error number defined by you.

The error is handled by the error processing mechanism currently in use – either OPL's own, which stops the program and displays an error message, or the ONERR handler if you have ONERR on.

RANDOMIZE Seeds the random number generator

Usage: RANDOMIZE x&

Gives a 'seed' (start value) for RND.

Successive calls of the RND function produce a sequence of pseudo-random numbers. If you use RANDOMIZE to set the seed back to what it was at the beginning of the sequence, the same sequence will be repeated.

For example, you might want to use the same 'random' values to test new versions of a procedure. To do this, precede the RND statement with the statement RANDOMIZE value. Then to repeat the sequence, use RANDOMIZE value again.

Example:

```
PROC SEQ:
  LOCAL g$(1)
  WHILE 1
    PRINT "S: set seed to 1"
    PRINT "Q: quit"
    PRINT "other key: continue"
    g$=UPPER$(GET$)
    IF g$="Q"
      BREAK
    ELSEIF g$="S"
      PRINT "Setting seed to 1"
      RANDOMIZE 1
      PRINT "First random no:"
    ELSE
      PRINT "Next random no:"
    ENDIF
    PRINT RND
```

```
    ENDWH
  ENDP
```

REALLOC — Changes the size of a previously allocated cell

Usage: pcelln&=REALLOC(pcell&,size&)

Change the size of a previously allocated cell at pcell& to size&, returning the new cell address or zero if there is not enough memory.

See also SETFLAGS if you require the 64K limit to be enforced. If the flag is set to restrict the limit, pcelln& is guaranteed to fit into an integer.

See also Dynamic memory allocation.

ARM

Cells are allocated lengths that are the smallest multiple of four greater than the size requested because the ARM processor requires a 4-byte word alignment for its memory allocation.

REM — Comment marker

Usage: REM text

Precedes a remark you include to explain how a program works. All text after the REM up to the end of the line is ignored.

When you use REM at the end of a line you need only precede it with a space, not a space and a colon.

Examples:

```
INPUT a :b=a*.175 REM b=TAX
INPUT a :b=a*.175 : REM b=TAX
```

RENAME — Renames files

Usage: RENAME file1$,file2$

Renames file1$ as file2$. You can rename any type of file.

You cannot use wildcards.

You can rename across directories: RENAME "\dat\xyz.abc","\xyz.abc" is OK. If you do this, you can choose whether or not to change the name of the file.

Example:

```
PRINT "Old name:" :INPUT a$
PRINT "New name:" :INPUT b$
RENAME a$,b$
```

REPT$ Repeats a string

Usage: r$=REPT$(a$,x%)

Returns a string comprising x% repetitions of a$.
For example, if a$="ex", r$=REPT$(a$,5) returns exexexexex.

RETURN Returns from a procedure

Usage: RETURN
or RETURN variable

Terminates the execution of a procedure and returns control to the point where that procedure was called (ENDP does this automatically).
RETURN variable does this as well, but also passes the value of variable back to the calling procedure. The variable may be of any type. You can return the value of any single array element – for example, RETURN x%(3). You can only return one variable.
RETURN on its own, and the default return through ENDP, causes the procedure to return the value 0 or a null string.

RIGHT$ Gets the rightmost characters of string

Usage: r$=RIGHT$(a$,x%)

Returns the rightmost x% characters of a$.
Example:

```
PRINT "Enter name/ref",
INPUT c$
ref$=RIGHT$(c$,4)
name$=LEFT$(c$,LEN(c)$-4)
```

ROLLBACK Cancels the current transaction on the current view

Usage: ROLLBACK

Cancels the current transaction on the current view. Changes made to the database with respect to this

particular view since BEGINTRANS was called will be discarded.

See also BEGINTRANS, COMMITTRANS.

RMDIR — Removes directories

Usage: RMDIR str$

Removes the directory given by str$. You can only remove empty directories.

RND — Gets a pseudo-random floating point number

Usage: r=RND

Returns a pseudo-random floating point number in the range 0 (inclusive) to 1 (exclusive).

To produce random numbers between 1 and n, e.g. between 1 and 6 for a dice, use the following statement: f%=1+INT(RND*n).

RND produces a different number every time it is called within a program. A fixed sequence can be generated by using RANDOMIZE. You might begin by using RANDOMIZE with an argument generated from MINUTE and SECOND (or similar), to seed the sequence differently each time.

Example:

```
PROC rndvals:
  LOCAL i%
  PRINT "Random test values:"
  DO
    PRINT RND
    i%=i%+1
    GET
  UNTIL i%=10
ENDP
```

SCI$ — Converts a number to scientific format

Usage: s$=SCI$(x,y%,z%)

Returns a string representation of x in scientific format, to y% decimal places and up to z% characters wide.

Examples:

```
SCI$(123456,2,8)="1.23E+05"
SCI$(1,2,8)="1.00E+00"
SCI$(1234567,1,-8)="1.2E+06"
```

If the number does not fit in the width specified then the returned string contains asterisks.

If z% is negative then the string is right-justified.

See also FIX$, GEN$, NUM$.

SCREEN Changes the size of the text window

Usage: SCREEN width%,height%
or SCREEN width%,height%,x%,y%

Changes the size of the window in which text is displayed. x%,y% specify the character position of the top left corner; if they are not given, the text window is centered in the screen.

An OPL program can initially display text to the whole screen.

See SCREENINFO.

SCREENINFO Gets information about the text screen

Usage: SCREENINFO var info%()

Gets information on the text screen (as used by PRINT, SCREEN, etc.).

This keyword allows you to mix text and graphics. It is required because while the default window is the same size as the physical screen, the text screen is slightly smaller and is centered in the default window. The few pixels gaps around the text screen, referred to as the left and top margins, depend on the font in use.

On return, info%() contains the information. info%() must have at least 10 elements. The information is returned at the following indices in info%():

KSinfoALeft%	1	left margin in pixels
KSinfoATop%	2	top margin in pixels
KSinfoAScrW%	3	text screen width in character units
KSinfoAScrH%	4	text screen height in character units
KSinfoAReserved1%	5	reserved (window server ID for default window)
KSinfoAFont%	6	unused (font ID for older systems)
KSinfoAPixW%	7	pixel width of text window character cell
KSinfoAPixH%	8	pixel height of text window character cell
KSinfoAReserved2%	9	least significant 16 bits of the font ID
KSinfoAReserved3%	10	most significant 16 bits of the font ID

These constants are supplied in Const.oph.

The font ID is a 32-bit integer under Symbian OS, and therefore would not fit into a single element of info%(). Hence, the least significant 16 bits of the font ID are returned to info%(9) and the most significant 16 bits to info%(10).

Initially SCREENINFO returns the values for the initial text screen. Subsequently, any keyword that changes the size of the text screen font, such as FONT or SCREEN, will change some of these values and SCREENINFO should therefore be called again.

See also FONT, SCREEN.

SECOND — Gets the current time in seconds

Usage: s%=SECOND

Returns the current time in seconds from the system clock (0 to 59).

E.g. at 6:00:33 SECOND returns 33.

SECSTODATE — Converts seconds to date

Usage: SECSTODATE s&,var yr%,var mo%,var dy%,var hr%,var mn%, var sc%,var yrday%

Sets the variables passed by reference to the date corresponding to s&, the number of seconds since 00:00 on 1/1/1970. yrday% is set to the day in the year (1–366).

s& is an unsigned long integer. To use values greater than +2147483647, subtract 4294967296 from the value.

See also DATETOSECS, HOUR, MINUTE, SECOND, dDATE, DAYS.

SETDOC — Sets a file to be a document

Usage: SETDOC file$

Sets the file file$ to be a document. This command should be called immediately before the creation of file$ if it is to be recognized as a document. SETDOC may be used with the commands CREATE, gSAVEBIT, and IOOPEN.

The string passed to SETDOC must be identical to the name passed to the following CREATE or gSAVEBIT otherwise a non-document file will be created. Example of document creation:

```
SETDOC "myfile"
CREATE "myfile",a,a$,b$
```

SETDOC should also be called after successfully opening a document to allow the System screen to display the correct document name in its task list.

In case of failure in creating or opening the required file, you should take the following action:

Creating – try to re-open the last file and if this fails display an appropriate error dialog and exit. On reopening, call SETDOC back to the original file so the Task list is correct.

Opening – as for creating, but calling SETDOC again is not strictly required.

Database documents, created using CREATE, and multi-bitmap documents, created using gSAVEBIT, will automatically contain your application UID in the file header. For binary and text file documents created using IOOPEN and LOPEN, it is the programmer's responsibility to save the appropriate header in the file. This is a fairly straightforward process and the following suggests one way of finding out what the header should be:

Create a database or bitmap document in a test run of your application using SETDOC as shown above.

Use a hex editor or hex dump program to find the first 16 bytes, or run the program below which reads the four long integer UIDs from the test document.

Write these four long integers to the start of the file you create using IOOPEN.

```
INCLUDE "Const.oph"
DECLARE EXTERNAL
EXTERNAL readUids:(file$)

PROC main:
  LOCAL f$(255)
  WHILE 1
    dINIT "Show UIDs in document header"
    dPOSITION 1,0
    dFILE f$,"Document,Folder,Drive",0
    IF DIALOG=0
      RETURN
    ENDIF
```

```
    readUids:(f$)
  ENDWH
ENDP

PROC readUids:(f$)
  LOCAL ret%,h%
  LOCAL uid&(4),i%
  ret%=IOOPEN(h%,f$,KIoOpenModeOpen% OR
      KIoOpenFormatBinary%)
  IF ret%>=0
    ret%=IOREAD(h%,ADDR(uid&()),16)
    PRINT "Reading ";f$
    IF ret%=16
      WHILE i%<4
        i%=i%+1
       PRINT " Uid"+num$(i%,1)+"=",hex$(uid&(i%))
      ENDWH
    ELSE
      PRINT " Error reading: ";
      IF ret%<0
        PRINT err$(ret%)
      ELSE
        PRINT "Read ";ret%;" bytes only ";
        PRINT "(4 long integers required)"
      ENDIF
    ENDIF
    IOCLOSE(h%)
  ELSE
    PRINT "Error opening: ";ERR$(ret%)
  ENDIF
ENDP
```

Creating text file documents using IOOPEN or LOPEN has two special requirements:

You will need to save the required header as the first text record. This will insert the standard text file line delimiters CR LF (hex 0D 0A) at the end of the record.

The specific 16 bytes required for your application may itself however contain CR LF. Since you should know when this is the case, you will need to read records until you have reached byte 16 in the document. This is clearly not a desirable state of affairs, but is inescapable given that text files were not designed to have headers. It is recommended that you request a new UID for your application if it contains CR LF.

See also GETDOC$.

SETFLAGS Sets an application's flags

Usage: SETFLAGS flags&

Sets flags to produce various effects when running programs. Use CLEARFLAGS to clear any flags that have been set. flags& is formed by adding one or more of the following values:

KRestrictTo64K& 1

restricts the memory available to your application to 64K, emulating an older 16-bit machine. This setting should be used at the beginning of your program only, if required. Changing this setting repeatedly will have unpredictable effects.

KAutoCompact& 2

enables auto-compaction on closing databases. This can be slow, but it is advisable to use this setting when lots of changes have been made to a database.

KTwoDigitExponent& 4

enables raising of overflow errors when floating point values are greater than or equal to 1.0E+100 in magnitude, instead of allowing three-digit exponents (for backwards compatibility).

KSendSwitchOnMessage& $10000

enables GETEVENT, GETEVENT32, and GETEVENT32A to return the event code $403 to ev&(1) when the machine switches on.

These constants are supplied in Const.oph.

By default these flags are cleared.

See also GETEVENT32, CLEARFLAGS.

SETHELP Sets the help context to be displayed

Usage: SETHELP ViewType%,HelpText$

Sets the current help context. For example, to show the help for the current view/state of your application (in the main event-handler):

```
IF (Key&=KKeyHelp32&) AND (Mods&=0)
   IF View%=KMainView%
      SETHELP KHelpView%,KMyHelpMainView$
   ELSEIF View%=KOtherView%
      SETHELP KHelpView%,KMyHelpOtherView$
   ELSE
      SETHELP KHelpView%,KMyHelpQuickStart$
   ENDIF
   SHOWHELP
ENDIF
```

To ensure that a given topic is displayed if the Help key is pressed while a dialog is on display (you cannot 'trap' the key itself while the dialog is on show – OPL does this for you):

```
SETHELP KHelpDialog%,KMyHelpSettingsDialog$
dINIT "Settings"
..other dialog controls, etc. as normal..
Dia%=DIALOG
```

To ensure that a given topic is displayed if the Help key is pressed while a menu is on display (again, OPL will 'trap' the key for you):

```
SETHELP KHelpMenu%,KMyHelpMainMenu$
mINIT
mCARD "File",..etc..
..other menu controls, etc. as normal..
Menu&=MENU(LastMenuValue%)
```

See also SETHELPUID, SHOWHELP.

SETHELPUID Sets the help file to be displayed by your application

Usage: SETHELP KHelpFileUID$

Sets the help file to be displayed by your application, using the unique identifier of that help file. This UID would normally be the same as that of your own application. It can be set on the PC using the SDK at the time of building the help file.

See also SETHELP, SHOWHELP.

SETPATH Sets the path for file access

Usage: SETPATH name$

Sets the current path for file access. For example:

SETPATH "C:\Documents\"

SETPATH needs the final backslash to be passed otherwise it ignores everything beyond the last backslash. LOADM continues to use the path of the initial program, but all other file access will use the new path.

SHOWHELP Shows the help file

Usage: SHOWHELP

Displays the help file currently associated with your application, using the context most recently set by SETHELP.

See also SETHELPUID, SETHELP.

SIN Sine

Usage: s=SIN(angle)

Returns the sine of angle, an angle expressed in radians.

To convert from degrees to radians, use the RAD function.

SIZE Size of a string

Usage: S%=SIZE(String$)

Returns the size of string, not just its length (to allow for the fact that under Symbian OS all strings are stored in Unicode format). For example, the following code demonstrates the difference between the SIZE and LEN keywords:

```
PROC Main:
LOCAL String$(13)
        String$="How Big Am I?"
        PRINT "Length =",LEN(String$)
        PRINT "Size =",SIZE(String$)
        PRINT "Press any key to continue."
        GET
ENDP
```

This should result in the following console output:

Length = 13
Size = 26
Press any key to continue

See also LEN.

SPACE — Gets space available on the current file's device

Usage: s&=SPACE

Returns the number of free bytes on the device on which the current (open) data file is held.

SQR — Square root

Usage: s=SQR(x)

Returns the square root of x.

STD — Standard deviation

Usage: s=STD(list)
or s=STD(array(),element)

Returns the standard deviation of a list of numeric items.

The list can be either:
A list of variables, values, and expressions, separated by commas

or

The elements of a floating point array.
When operating on an array, the first argument must be the array name followed by (). The second argument, separated from the first by a comma, is the number of array elements you wish to operate on. For example, m=STD(arr(),3) would return the standard deviation of elements arr(1), arr(2), and arr(3).

STOP — Stops the running program

Usage: STOP

Ends the running program.
Note that STOP may not be used during an OPX callback and will raise the error 'STOP used in callback' if it is. See Callbacks from OPX procedures.

STYLE — Sets the text window character style

Usage: STYLE style%

Sets the text window character style. style% can be 2 for underlined, or 4 for inverse.

SUM — Sums a list of numeric items

Usage: s=SUM(list)
or s=SUM(array(),element)

Returns the sum of a list of numeric items.
The list can be either:
A list of variables, values, and expressions, separated by commas

or

The elements of a floating point array.
When operating on an array, the first argument must be the array name followed by (). The second argument, separated from the first by a comma, is the number of array elements you wish to operate on. For example, m=SUM(arr(),3) would return the sum of elements arr(1), arr(2), and arr(3).

TAN — Tangent

Usage: t=TAN(angle)

Returns the tangent of angle, an angle expressed in radians.
To convert from radians to degrees, use the DEG function.

TESTEVENT — Tests if an event has occurred

Usage: t%=TESTEVENT

Returns 'True' if an event has occurred, otherwise returns 'False'. The event is not read by TESTEVENT – it may be read with GETEVENT, GETEVENT32, or GETEVENTA32.

Warning: It is recommended that you use either GETEVENT32 or GETEVENTA32 without TESTEVENT as TESTEVENT may use a lot of power, especially when used in a loop as will often be the case.

TRAP — Traps errors

Section Contents

Trapping data file commands
Trapping file commands
Trapping directory commands
Trapping data entry commands
Trapping graphics commands

Usage: TRAP command

TRAP is an error handling command. It may precede any of these commands:

Trapping data file commands
APPEND, UPDATE, BACK, NEXT, LAST, FIRST, POSITION, USE, CREATE, OPEN, OPENR, CLOSE, DELETE, MODIFY, INSERT, PUT, CANCEL

Trapping file commands
COPY, ERASE, RENAME, LOPEN, LCLOSE, LOADM, UNLOADM

Trapping directory commands
MKDIR, RMDIR

Trapping data entry commands
EDIT, INPUT

Trapping graphics commands
gSAVEBIT, gCLOSE, gUSE, gUNLOADFONT, gFONT, gPATT, gCOPY

For example, TRAP FIRST.

Any error resulting from the execution of the command will be trapped. Program execution will continue at the statement after the TRAP statement, but ERR will be set to the error code.

TRAP overrides any ONERR.

TRAP RAISE — Clears the trap flag

Usage: TRAP RAISE x%

Sets the value of ERR to x% and clears the trap flag.

UADD — Adds two unsigned integers

Usage: i%=UADD(val1%, val2%)

Add val1% and val2%, as if both were unsigned integers with values from 0 to 65535. Prevents integer overflow for pointer arithmetic when the 64K memory restriction is set (see SETFLAGS), e.g. UADD(ADDR(text$),1) should be used instead of ADDR(text$)+1.

One argument would normally be a pointer and the other an offset expression.

Note that UADD and USUB should not be used for pointer arithmetic unless SETFLAGS has been used to enforce the 64K memory limit. In general, long integer arithmetic should be used for pointer arithmetic.

See also USUB.

UNLOADM Unloads a module

Usage: UNLOADM module$

Removes from memory the module module$ loaded with LOADM.

module$ is a string containing the name of the translated module.

The procedures in an unloaded module cannot then be called by another procedure.

UNLOADM causes any procedures in the module that are not still running to be unloaded from memory too. Running procedures are unloaded on return. It is considered bad practice, however, to use UNLOADM on a module with procedures that are still running.

Once LOADM has been called, procedures loaded stay in memory until the module is unloaded. Modules are not flushed automatically.

UNTIL See DO

See DO.

PDATE Deletes the current record and saves as a new record at the end of the file

Usage: UPDATE

Warning: This function is deprecated and included only for compatibility with older versions of the OPL language. INSERT, PUT, and CANCEL should be used in preference to APPEND and UPDATE, although APPEND and UPDATE are still supported. However, note that APPEND can generate a lot of extra (intermediate) erased records. COMPACT should be used to remove them, or alternatively use SETFLAGS to set auto-compaction on.

Deletes the current record in the current data file and saves the current field values as a new record at the end of the file.

This record, now the last in the file, remains the current record.

Example:

```
A.count=129
A.name$="Brown"
UPDATE
```

Use APPEND to save the current field values as a new record.

UPPER$ Converts a string to uppercase

Usage: u$=UPPER$(a$)

Converts any lowercase characters in a$ to uppercase, and returns the completely uppercase string. Example:

```
...
CLS :PRINT "Y to continue"
PRINT "or N to stop."
g$=UPPER$(GET$)
IF g$="Y"
  nextproc:
ELSEIF g$="N"
  RETURN
ENDIF
...
```

Use LOWER$ to convert to lowercase.

USE Selects a data file

Usage: USE logical name

Selects the data file with the given logical name (A–Z). The file must previously have been opened with OPEN, OPENR, or CREATE, and not yet be closed.

All the record handling commands (such as POSITION and UPDATE, and GOTOMARK, INSERT, MODIFY, CANCEL and PUT) then operate on this file.

USUB Subtracts two unsigned integers

Usage: i%=USUB(val1%,val2%)

Subtract val2% from val1%, as if both were unsigned integers with values from 0 to 65535. Prevents integer

overflow for pointer arithmetic when the 64K memory restriction is set (see SETFLAGS).

Note that UADD and USUB should not be used for pointer arithmetic unless SETFLAGS has been used to enforce a 64K memory limit. In general long integer arithmetic should be used.

See also UADD.

VAL — Converts numeric string to floating point number

Usage: v=VAL(numeric string)

Returns the floating point number corresponding to a numeric string.

The string must be a valid number, e.g. not "5.6.7" or "196f". Expressions such as "45.6*3.1" are not allowed. Scientific notation such as "1.3E10" is OK.

E.g. VAL("470.0") returns 470.

See also EVAL.

VAR — Gets variance of a list of items

Usage: v=VAR(list)
or v=VAR(array(),element)

Returns the variance of a list of numeric items.

The list can be either:

A list of variables, values, and expressions, separated by commas

or

The elements of a floating point array.

When operating on an array, the first argument must be the array name followed by (). The second argument, separated from the first by a comma, is the number of array elements you wish to operate on. For example, m=VAR(arr(),3) would return the variance of elements arr(1), arr(2), and arr(3).

This function gives the sample variance.

VECTOR — Jumps to a numbered label

Usage:

```
VECTOR i%
  label1,label2,...,labelN
ENDV
```

VECTOR i% jumps to label number i% in the list. If i% is 1 this will be the first label, and so on. The list is terminated by the ENDV statement. The list may spread over several lines, with a comma separating labels in any one line but no comma at the end of each line.

If i% is not in the range 1 to N, where N is the number of labels, the program continues with the statement after the ENDV statement.

See also GOTO.

WEEK — Gets the week in which a specified day falls

Usage: w%=WEEK(day%,month%,year%)

Returns the week number in which the specified day falls, as an integer between 1 and 53.

day% must be between 1 and 31, month% between 1 and 12, year% between 0 and 9999.

Each week is taken to begin on the 'Start of week' day, as specified in the Control Panel. When a year begins on a different day to the start of the week, it counts as week 1 if there are four or more days before the next week starts.

The System setting of the 'Start of week' may be checked from inside OPL by using the LCSTARTOFWEEK&: procedure in the Date OPX. The week number in the year may also be calculated by different rules and also with your own choice of the start of year by using the procedure DTWEEKNOINYEAR&: in Date OPX.

HILE...ENDWH — Conditional loop

Usage:

```
WHILE expression
  ...
ENDWH
```

Repeatedly performs the set of instructions between the WHILE and the ENDWH statement, so long as expression returns true (non-zero).

If expression is not true, the program jumps to the line after the ENDWH statement.

Every WHILE must be closed with a matching ENDWH.

See also DO...UNTIL.

YEAR — Gets the current year

Usage: y%=YEAR

Returns the current year as an integer from the system clock.

For example, on 5th May 1997 YEAR returns 1997.

Appendix 2

Const.oph Listing

```
rem CONST.OPH 6.01
rem Constants for OPL - Last updated 31 May 2004

rem
rem GENERAL CONSTANTS
rem

CONST KTrue%=-1
CONST KFalse%=0

rem Data type ranges
CONST KMaxStringLen%=255
CONST KMaxFloat=1.7976931348623157E+308
CONST KMinFloat=2.2250738585072015E-308 rem Minimum with full precision in
    mantissa
CONST KMinFloatDenorm=5e-324 rem Denormalised (just one bit of precision
    left)
CONST KMinInt%=$8000 rem -32768 (translator needs hex for maximum ints)
CONST KMaxInt%=32767
CONST KMinLong&=&80000000 rem -2147483648 (hex for translator)
CONST KMaxLong&=2147483647
CONST KMaxdTIMEValue&=86399

rem Data type sizes
CONST KShortIntWidth&=2
CONST KLongIntWidth&=4
CONST KFloatWidth&=8

rem Error codes
CONST KErrNone%=0
CONST KErrGenFail%=-1
CONST KErrInvalidArgs%=-2
CONST KErrOs%=-3
CONST KErrNotSupported%=-4
CONST KErrUnderflow%=-5
CONST KErrOverflow%=-6
CONST KErrOutOfRange%=-7
CONST KErrDivideByZero%=-8
CONST KErrInUse%=-9
CONST KErrNoMemory%=-10
CONST KErrNoSegments%=-11
CONST KErrNoSemaphore%=-12
CONST KErrNoProcess%=-13
CONST KErrAlreadyOpen%=-14
```

```
CONST KErrNotOpen%=-15
CONST KErrImage%=-16
CONST KErrNoReceiver%=-17
CONST KErrNoDevices%=-18
CONST KErrNoFileSystem%=-19
CONST KErrFailedToStart%=-20
CONST KErrFontNotLoaded%=-21
CONST KErrTooWide%=-22
CONST KErrTooManyItems%=-23
CONST KErrBatLowSound%=-24
CONST KErrBatLowFlash%=-25
CONST KErrExists%=-32
CONST KErrNotExists%=-33
CONST KErrWrite%=-34
CONST KErrRead%=-35
CONST KErrEof%=-36
CONST KErrFull%=-37
CONST KErrName%=-38
CONST KErrAccess%=-39
CONST KErrLocked%=-40
CONST KErrDevNotExist%=-41
CONST KErrDir%=-42
CONST KErrRecord%=-43
CONST KErrReadOnly%=-44
CONST KErrInvalidIO%=-45
CONST KErrFilePending%=-46
CONST KErrVolume%=-47
CONST KErrIOCancelled%=-48
rem OPL specific errors
CONST KErrSyntax%=-77
CONST KOplStructure%=-85
CONST KErrIllegal%=-96
CONST KErrNumArg%=-97
CONST KErrUndef%=-98
CONST KErrNoProc%=-99
CONST KErrNoFld%=-100
CONST KErrOpen%=-101
CONST KErrClosed%=-102
CONST KErrRecSize%=-103
CONST KErrModLoad%=-104
CONST KErrMaxLoad%=-105
CONST KErrNoMod%=-106
CONST KErrNewVer%=-107
CONST KErrModNotLoaded%=-108
CONST KErrBadFileType%=-109
CONST KErrTypeViol%=-110
CONST KErrSubs%=-111
CONST KErrStrTooLong%=-112
CONST KErrDevOpen%=-113
CONST KErrEsc%=-114
CONST KErrMaxDraw%=-117
CONST KErrDrawNotOpen%=-118
CONST KErrInvalidWindow%=-119
CONST KErrScreenDenied%=-120
CONST KErrOpxNotFound%=-121
CONST KErrOpxVersion%=-122
CONST KErrOpxProcNotFound%=-123
CONST KErrStopInCallback%=-124
CONST KErrIncompUpdateMode%=-125
CONST KErrInTransaction%=-126
```

```
rem -127 to -133 translator errors
CONST KErrBadAlignment%=-134

rem Month numbers
CONST KJanuary%=1
CONST KFebruary%=2
CONST KMarch%=3
CONST KApril%=4
CONST KMay%=5
CONST KJune%=6
CONST KJuly%=7
CONST KAugust%=8
CONST KSeptember%=9
CONST KOctober%=10
CONST KNovember%=11
CONST KDecember%=12

rem For DOW
CONST KMonday%=1
CONST KTuesday%=2
CONST KWednesday%=3
CONST KThursday%=4
CONST KFriday%=5
CONST KSaturday%=6
CONST KSunday%=7

rem DATIM$ offsets
CONST KDatimOffDayName%=1
CONST KDatimOffDay%=5
CONST KDatimOffMonth%=8
CONST KDatimOffYear%=12
CONST KDatimOffHour%=17
CONST KDatimOffMinute%=20
CONST KDatimOffSecond%=23

rem Help location values
CONST KHelpView%=0
CONST KHelpDialog%=1
CONST KHelpMenu%=2

rem For BUSY and GIPRINT
CONST KBusyTopLeft%=0
CONST KBusyBottomLeft%=1
CONST KBusyTopRight%=2
CONST KBusyBottomRight%=3
CONST KBusyMaxText%=80

rem For CMD$
CONST KCmdAppName%=1 rem Full path name used to start program
CONST KCmdUsedFile%=2
CONST KCmdLetter%=3
rem For CMD$(3)
CONST KCmdLetterCreate$="C"
CONST KCmdLetterOpen$="O"
CONST KCmdLetterRun$="R"
CONST KCmdLetterBackground$="B"
CONST KCmdLetterViewActivate$="V"
CONST KCmdLetterRunWithoutViews$="W"

rem For GETCMD$
```

```
CONST KGetCmdLetterCreate$="C"
CONST KGetCmdLetterOpen$="O"
CONST KGetCmdLetterExit$="X"
CONST KGetCmdLetterBroughtToFGround$="F"
CONST KGetCmdLetterBackup$="S"
CONST KGetCmdLetterRestart$="R"
CONST KGetCmdLetterUnknown$="U"

rem PARSE$ array subscripts
CONST KParseAOffFSys%=1
CONST KParseAOffDev%=2
CONST KParseAOffPath%=3
CONST KParseAOffFilename%=4
CONST KParseAOffExt%=5
CONST KParseAOffWild%=6
rem Wild-card flags
CONST KParseWildNone%=0
CONST KParseWildFilename%=1
CONST KParseWildExt%=2
CONST KParseWildBoth%=3

rem For CURSOR
CONST KCursorTypeNotFlashing%=2
CONST KCursorTypeGray%=4

rem For FINDFIELD
CONST KFindCaseDependent%=16
CONST KFindBackwards%=0
CONST KFindForwards%=1
CONST KFindBackwardsFromEnd%=2
CONST KFindForwardsFromStart%=3

rem SCREENINFO array subscripts
CONST KSInfoALeft%=1
CONST KSInfoATop%=2
CONST KSInfoAScrW%=3
CONST KSInfoAScrH%=4
CONST KSInfoAReserved1%=5
CONST KSInfoAFont%=6
CONST KSInfoAPixW%=7
CONST KSInfoAPixH%=8
CONST KSInfoAReserved2%=9
CONST KSInfoAReserved3%=10

rem Unicode ellipsis, linefeed and carriage-return
CONST KEllipsis&=&2026
CONST KLineFeed&=10
CONST KCarriageReturn&=13

rem For SETFLAGS
CONST KRestrictTo64K&=&0001
CONST KAutoCompact&=&0002
CONST KTwoDigitExponent&=&0004
CONST KMenuCancelCompatibility&=&0008
CONST KAlwaysWriteAsciiTextFiles&=&0016
CONST KSendSwitchOnMessage&=&10000

rem To aid porting to Unicode OPL
CONST KOplAlignment%=1
CONST KOplStringSizeFactor%=2
```

```
rem
rem EVENT HANDLING
rem

rem Special keys
CONST KKeyDel%=8
CONST KKeyTab%=9
CONST KKeyEnter%=13
CONST KKeyEsc%=27
CONST KKeySpace%=32

rem Scan code values
CONST KScanDel%=1
CONST KScanTab%=2
CONST KScanEnter%=3
CONST KScanEsc%=4
CONST KScanSpace%=5

rem GETEVENT32 array indexes
CONST KEvAType%=1
CONST KEvATime%=2
CONST KEvAScan%=3
CONST KEvAKMod%=4
CONST KEvAKRep%=5

rem Pointer event array subscripts
CONST KEvAPtrOplWindowId%=3
CONST KEvAPtrWindowId%=3
CONST KEvAPtrType%=4
CONST KEvAPtrModifiers%=5
CONST KEvAPtrPositionX%=6
CONST KEvAPtrPositionY%=7
CONST KEvAPtrScreenPosX%=8
CONST KEvAPtrScreenPosY%=9

rem Event types
CONST KEvNotKeyMask&=&400
CONST KEvFocusGained&=&401
CONST KEvFocusLost&=&402
CONST KEvSwitchOn&=&403
CONST KEvCommand&=&404
CONST KEvDateChanged&=&405
CONST KEvKeyDown&=&406
CONST KEvKeyUp&=&407
CONST KEvPtr&=&408
CONST KEvPtrEnter&=&409
CONST KEvPtrExit&=&40A

rem Pointer event types
CONST KEvPtrPenDown&=0
CONST KEvPtrPenUp&=1
CONST KEvPtrButton1Down&=KEvPtrPenDown&
CONST KEvPtrButton1Up&=KEvPtrPenUp&
CONST KEvPtrButton2Down&=2
CONST KEvPtrButton2Up&=3
CONST KEvPtrButton3Down&=4
CONST KEvPtrButton3Up&=5
CONST KEvPtrDrag&=6
CONST KEvPtrMove&=7
CONST KEvPtrButtonRepeat&=8
```

```
CONST KEvPtrSwitchOn&=9

rem For PointerFilter
CONST KPointerFilterEnterExit%=$1
CONST KPointerFilterMove%=$2
CONST KPointerFilterDrag%=$4

rem Key constants (for 32-bit keywords like GETEVENT32)
CONST KKeyHelp32&=&f83a
CONST KKeyMenu32&=&f836
CONST KKeySidebarMenu32&=&f700
CONST KKeyPageLeft32&=&f802
CONST KKeyPageRight32&=&f803
CONST KKeyPageUp32&=&f804
CONST KKeyPageDown32&=&f805
CONST KKeyLeftArrow32&=&f807
CONST KKeyRightArrow32&=&f808
CONST KKeyUpArrow32&=&f809
CONST KKeyDownArrow32&=&f80a
rem For the command button array
CONST KKeyCBA1&=&f842
CONST KKeyCBA2&=&f843
CONST KKeyCBA3&=&f844
CONST KKeyCBA4&=&f845
rem Special keys
CONST KKeyZoomIn32&=&f703
CONST KKeyZoomOut32&=&f704
CONST KKeyIncBrightness32&=&f864

rem For 32-bit status words IOWAIT and IOWAITSTAT32
rem Use KErrFilePending% (-46) for 16-bit status words
CONST KStatusPending32&=&80000001

rem For KMOD
CONST KKmodShift%=2
CONST KKmodControl%=4
CONST KKmodCaps%=16
CONST KKmodFn%=32

rem
rem DIALOGS
rem

rem For ALERT
CONST KAlertEsc%=1
CONST KAlertEnter%=2
CONST KAlertSpace%=3

rem For dBUTTON
CONST KDButtonNoLabel%=$100
CONST KDButtonPlainKey%=$200
CONST KDButtonBlank$=""
CONST KDButtonBlank%=0
CONST KDButtonDel%=8
CONST KDButtonTab%=9
CONST KDButtonEnter%=13
CONST KDButtonEsc%=27
CONST KDButtonSpace%=32
rem DIALOG return values
CONST KDlgCancel%=0
```

```
rem For dEDITMULTI and printing
CONST KParagraphDelimiter&=$2029 rem $06 under ASCII
CONST KLineBreak&=$2028                  rem $07 under ASCII
CONST KPageBreak&=$000c                  rem $08 under ASCII
CONST KTabCharacter&=$0009       rem $09 under ASCII
CONST KNonBreakingHyphen&=$2011          rem $0b under ASCII
CONST KPotentialHyphen&=$00ad            rem $0c under ASCII
CONST KNonBreakingSpace&=$00a0           rem $10 under ASCII
CONST KPictureCharacter&=$fffc           rem $0e under ASCII
CONST KVisibleSpaceCharacter&=$0020      rem $0f under ASCII

rem For dFILE
CONST KDFileNameLen%=255
rem flags
CONST KDFileEditBox%=$0001
CONST KDFileAllowFolders%=$0002
CONST KDFileFoldersOnly%=$0004
CONST KDFileEditorDisallowExisting%=$0008
CONST KDFileEditorQueryExisting%=$0010
CONST KDFileAllowNullStrings%=$0020
CONST KDFileAllowWildCards%=$0080
CONST KDFileSelectorWithRom%=$0100
CONST KDFileSelectorWithSystem%=$0200
CONST KDFileSelectorAllowNewFolder%=$0400
CONST KDFileSelectorShowHidden%=$0800

rem Current OPL-related UIDs (for dFILE UID restriction)
CONST KUidDirectFileStore&=&10000037
CONST KUidOplInterpreter&=&10005D2E
CONST KUidOpo&=&100055C0
CONST KUidOplApp&=&100055C1
CONST KUidOplDoc&=&100055C2
CONST KUidOplFile&=&1000008A
CONST KUidOpxDll&=&10003A7B

rem dINIT flags
CONST KDlgButRight%=1
CONST KDlgNoTitle%=2
CONST KDlgFillScreen%=4
CONST KDlgNoDrag%=8
CONST KDlgDensePack%=16

rem For dPOSITION
CONST KDPositionLeft%=-1
CONST KDPositionCenter%=0
CONST KDPositionRight%=1
CONST KDPositionTop%=-1
CONST KDPositionBottom%=1

rem For dTEXT
CONST KDTextLeft%=0
CONST KDTextRight%=1
CONST KDTextCenter%=2
CONST KDTextBold%=$100 rem Currently ignored
CONST KDTextLineBelow%=$200
CONST KDTextAllowSelection%=$400
CONST KDTextSeparator%=$800
rem For dTIME
CONST KDTimeAbsNoSecs%=0
CONST KDTimeAbsWithSecs%=1
```

```
CONST KDTimeDurationNoSecs%=2
CONST KDTimeDurationWithSecs%=3
rem Flags for dTIME (for ORing combinations)
CONST KDTimeWithSeconds%=1
CONST KDTimeDuration%=2
CONST KDTimeNoHours%=4
CONST KDTime24Hour%=8

rem For dXINPUT
CONST KDXInputMaxLen%=32

rem For Standard No/Yes dCHOICEs
CONST KNoYesChoiceNo%=1
CONST KNoYesChoiceYes%=2

rem
rem MENUS
rem

rem For mCARD and mCASC
CONST KMenuDimmed%=$1000
CONST KMenuSymbolOn%=$2000
CONST KMenuSymbolIndeterminate%=$4000
CONST KMenuCheckBox%=$0800
CONST KMenuOptionStart%=$0900
CONST KMenuOptionMiddle%=$0a00
CONST KMenuOptionEnd%=$0b00

rem mPOPUP position type - Specifies which corner
rem of the popup is given by supplied coordinates
CONST KMPopupPosTopLeft%=0
CONST KMPopupPosTopRight%=1
CONST KMPopupPosBottomLeft%=2
CONST KMPopupPosBottomRight%=3

rem
rem GRAPHICS
rem

rem For DEFAULTWIN
CONST KDefaultWin2GrayMode%=0
CONST KDefaultWin4GrayMode%=1
CONST KDefaultWin16GrayMode%=2
CONST KDefaultWin256GrayMode%=3
CONST KDefaultWin16ColorMode%=4
CONST KDefaultWin256ColorMode%=5
CONST KDefaultWin64KMode%=6
CONST KDefaultWin16MMode%=7
CONST KDefaultWinRGBMode%=8
CONST KDefaultWin4KMode%=9

CONST KDefaultWin%=1
CONST KgModeSet%=0
CONST KgModeClear%=1
CONST KgModeInvert%=2
CONST KtModeSet%=0
CONST KtModeClear%=1
CONST KtModeInvert%=2
CONST KtModeReplace%=3
```

```
CONST KgStyleNormal%=0
CONST KgStyleBold%=1
CONST KgStyleUnder%=2
CONST KgStyleInverse%=4
CONST KgStyleDoubleHeight%=8
CONST KgStyleMonoFont%=16
CONST KgStyleItalic%=32

rem RGB color masking
CONST KRgbRedPosition&=&10000
CONST KRgbGreenPosition&=$100
CONST KRgbBluePosition&=$1
CONST KRgbColorMask&=$ff

rem RGB color values
CONST KRgbBlack&=&000000
CONST KRgbDarkGray&=&555555
CONST KRgbDarkRed&=&800000
CONST KRgbDarkGreen&=&008000
CONST KRgbDarkYellow&=&808000
CONST KRgbDarkBlue&=&000080
CONST KRgbDarkMagenta&=&800080
CONST KRgbDarkCyan&=&008080
CONST KRgbRed&=&ff0000
CONST KRgbGreen&=&00ff00
CONST KRgbYellow&=&ffff00
CONST KRgbBlue&=&0000ff
CONST KRgbMagenta&=&ff00ff
CONST KRgbCyan&=&00ffff
CONST KRgbGray&=&aaaaaa
CONST KRgbDitheredLightGray&=&cccccc
CONST KRgb1in4DitheredGray&=&ededed
CONST KRgbWhite&=&ffffff

rem Easy mappings to the above RGB color combinations
CONST KColorSettingBlack%=1
CONST KColorSettingDarkGrey%=2
CONST KColorSettingDarkRed%=3
CONST KColorSettingDarkGreen%=4
CONST KColorSettingDarkYellow%=5
CONST KColorSettingDarkBlue%=6
CONST KColorSettingDarkMagenta%=7
CONST KColorSettingDarkCyan%=8
CONST KColorSettingRed%=9
CONST KColorSettingGreen%=10
CONST KColorSettingYellow%=11
CONST KColorSettingBlue%=12
CONST KColorSettingMagenta%=13
CONST KColorSettingCyan%=14
CONST KColorSettingGrey%=15
CONST KColorSettingLightGrey%=16
CONST KColorSettingLighterGrey%=17
CONST KColorSettingWhite%=18

rem For gBORDER and gXBORDER
CONST KBordSglShadow%=1
CONST KBordSglGap%=2
CONST KBordDblShadow%=3
CONST KBordDblGap%=4
CONST KBordGapAllRound%=$100
```

```
CONST KBordRoundCorners%=$200
CONST KBordLosePixel%=$400

rem For gBUTTON
CONST KButtSinglePixel%=0
CONST KButtSinglePixelRaised%=0
CONST KButtSinglePixelPressed%=1
CONST KButtDoublePixel%=1
CONST KButtDoublePixelRaised%=0
CONST KButtDoublePixelSemiPressed%=1
CONST KButtDoublePixelSunken%=2
CONST KButtStandard%=2
CONST KButtStandardRaised%=0
CONST KButtStandardSemiPressed%=1

CONST KButtLayoutTextRightPictureLeft%=0
CONST KButtLayoutTextBottomPictureTop%=1
CONST KButtLayoutTextTopPictureBottom%=2
CONST KButtLayoutTextLeftPictureRight%=3
CONST KButtTextRight%=0
CONST KButtTextBottom%=1
CONST KButtTextTop%=2
CONST KButtTextLeft%=3
CONST KButtExcessShare%=$00
CONST KButtExcessToText%=$10
CONST KButtExcessToPicture%=$20

rem For gCLOCK
CONST KClockLocaleConformant%=6
CONST KClockSystemSetting%=KClockLocaleConformant%
CONST KClockAnalog%=7
CONST KClockDigital%=8
CONST KClockLargeAnalog%=9
rem gClock 10 no longer supported (use slightly changed gCLOCK 11)
CONST KClockFormattedDigital%=11

rem For gCREATE
CONST KgCreateInvisible%=0
CONST KgCreateVisible%=1
CONST KgCreateHasShadow%=$0010
rem Color mode constants
CONST KgCreate2GrayMode%=$0000
CONST KgCreate4GrayMode%=$0001
CONST KgCreate16GrayMode%=$0002
CONST KgCreate256GrayMode%=$0003
CONST KgCreate16ColorMode%=$0004
CONST KgCreate256ColorMode%=$0005
CONST KgCreate64KColorMode%=$0006
CONST KgCreate16MColorMode%=$0007
CONST KgCreateRGBColorMode%=$0008
CONST KgCreate4KColorMode%=$0009

rem gCOLORINFO array subscripts
CONST gColorInfoADisplayMode%=1
CONST gColorInfoANumColors%=2
CONST gColorInfoANumGrays%=3
rem DisplayMode constants
CONST KDisplayModeNone%=0
CONST KDisplayModeGray2%=1
CONST KDisplayModeGray4%=2
```

```
CONST KDisplayModeGray16%=3
CONST KDisplayModeGray256%=4
CONST KDisplayModeColor16%=5
CONST KDisplayModeColor256%=6
CONST KDisplayModeColor64K%=7
CONST KDisplayModeColor16M%=8
CONST KDisplayModeRGB%=9
CONST KDisplayModeColor4K%=10

rem For gINFO
CONST KgInfoSize%=32
CONST KgInfoLowestCharCode%=1
CONST KgInfoHighestCharCode%=2
CONST KgInfoFontHeight%=3
CONST KgInfoFontDescent%=4
CONST KgInfoFontAscent%=5
CONST KgInfoWidth0Char%=6
CONST KgInfoMaxCharWidth%=7
CONST KgInfoFontFlag%=8
CONST KgInfoFontName%=9
rem 9-17 Font name
CONST KgInfogGMode%=18
CONST KgInfogTMode%=19
CONST KgInfogStyle%=20
CONST KgInfoCursorState%=21
CONST KgInfoCursorWindowId%=22
CONST KgInfoCursorWidth%=23
CONST KgInfoCursorHeight%=24
CONST KgInfoCursorAscent%=25
CONST KgInfoCursorX%=26
CONST KgInfoCursorY%=27
CONST KgInfoDrawableBitmap%=28
CONST KgInfoCursorEffects%=29
CONST KgInfogGray%=30
CONST KgInfoDrawableId%=31

rem For gINFO32
CONST KgInfo32Size%=48
rem 1,2 reserved
CONST KgInfo32FontHeight%=KgInfoFontHeight%
CONST KgInfo32FontDescent%=KgInfoFontDescent%
CONST KgInfo32FontAscent%=KgInfoFontAscent%
CONST KgInfo32Width0Char%=KgInfoWidth0Char%
CONST KgInfo32MaxCharWidth%=KgInfoMaxCharWidth%
CONST KgInfo32FontFlag%=KgInfoFontFlag%
CONST KgInfo32FontUID%=9
rem 10-17 unused
CONST KgInfo32gGMode%=KgInfogGMode%
CONST KgInfo32gTMode%=KgInfogTMode%
CONST KgInfo32gStyle%=KgInfogStyle%
CONST KgInfo32CursorState%=KgInfoCursorState%
CONST KgInfo32CursorWindowId%=KgInfoCursorWindowId%
CONST KgInfo32CursorWidth%=KgInfoCursorWidth%
CONST KgInfo32CursorHeight%=KgInfoCursorHeight%
CONST KgInfo32CursorAscent%=KgInfoCursorAscent%
CONST KgInfo32CursorX%=KgInfoCursorX%
CONST KgInfo32CursorY%=KgInfoCursorY%
CONST KgInfo32DrawableBitmap%=KgInfoDrawableBitmap%
CONST KgInfo32CursorEffects%=KgInfoCursorEffects%
CONST KgInfo32GraphicsMode%=30
```

```
CONST KgInfo32ForegroundRed%=31
CONST KgInfo32ForegroundGreen%=32
CONST KgInfo32ForegroundBlue%=33
CONST KgInfo32BackgroundRed%=34
CONST KgInfo32BackgroundGreen%=35
CONST KgInfo32BackgroundBlue%=36

rem For gLOADBIT
CONST KgLoadBitReadOnly%=0
CONST KgLoadBitWriteable%=1

rem For gRANK
CONST KgRankForeground%=1
CONST KgRankBackGround%=KMaxInt%

rem gPOLY array subscripts
CONST KgPolyAStartX%=1
CONST KgPolyAStartY%=2
CONST KgPolyANumPairs%=3
CONST KgPolyANumDx1%=4
CONST KgPolyANumDy1%=5

rem For gPRINTB
CONST KgPrintBRightAligned%=1
CONST KgPrintBLeftAligned%=2
CONST KgPrintBCenteredAligned%=3
rem The defaults
CONST KgPrintBDefAligned%=KgPrintBLeftAligned%
CONST KgPrintBDefTop%=0
CONST KgPrintBDefBottom%=0
CONST KgPrintBDefMargin%=0

rem For gXBORDER
CONST KgXBorderSinglePixelType%=0
CONST KgXBorderDoublePixelType%=1
CONST KgXBorderStandardType%=2

rem For gXPRINT
CONST KgXPrintNormal%=0
CONST KgXPrintInverse%=1
CONST KgXPrintInverseRound%=2
CONST KgXPrintThinInverse%=3
CONST KgXPrintThinInverseRound%=4
CONST KgXPrintUnderlined%=5
CONST KgXPrintThinUnderlined%=6

rem For gFONT
rem (Only suitable for devices using EON14.GDR e.g. Series 5, 9210)
CONST KFontArialBold8&=           268435951
CONST KFontArialBold11&=   268435952
CONST KFontArialBold13&=   268435953
CONST KFontArialNormal8&=  268435954
CONST KFontArialNormal11&= 268435955
CONST KFontArialNormal13&= 268435956
CONST KFontArialNormal15&= 268435957
CONST KFontArialNormal18&= 268435958
CONST KFontArialNormal22&= 268435959
CONST KFontArialNormal27&= 268435960
CONST KFontArialNormal32&= 268435961
```

```
CONST KFontTimesBold8&=           268435962
CONST KFontTimesBold11&=   268435963
CONST KFontTimesBold13&=   268435964
CONST KFontTimesNormal8&=  268435965
CONST KFontTimesNormal11&= 268435966
CONST KFontTimesNormal13&= 268435967
CONST KFontTimesNormal15&= 268435968
CONST KFontTimesNormal18&= 268435969
CONST KFontTimesNormal22&= 268435970
CONST KFontTimesNormal27&= 268435971
CONST KFontTimesNormal32&= 268435972

CONST KFontCourierBold8&=  268436062
CONST KFontCourierBold11&= 268436063
CONST KFontCourierBold13&= 268436064
CONST KFontCourierNormal8&=       268436065
CONST KFontCourierNormal11&=      268436066
CONST KFontCourierNormal13&=      268436067
CONST KFontCourierNormal15&=      268436068
CONST KFontCourierNormal18&=      268436069
CONST KFontCourierNormal22&=      268436070
CONST KFontCourierNormal27&=      268436071
CONST KFontCourierNormal32&=      268436072

CONST KFontCalc13n&=              268435493
CONST KFontCalc18n&=              268435494
CONST KFontCalc24n&=              268435495

CONST KFontMon18n&=      268435497
CONST KFontMon18b&=      268435498
CONST KFontMon9n&=       268435499
CONST KFontMon9b&=       268435500

CONST KFontTiny1&=       268435501
CONST KFontTiny2&=       268435502
CONST KFontTiny3&=       268435503
CONST KFontTiny4&=       268435504

CONST KFontEiksym15&=          268435661

CONST KFontSquashed&=          268435701
CONST KFontDigital35&=         268435752

rem
rem The following font consts are for Series 60 devices only
rem e.g. Nokia 7650
rem

CONST KFontS60LatinPlain12&=&10000001
CONST KFontS60LatinBold12&=&10000002
CONST KFontS60LatinBold13&=&10000003
CONST KFontS60LatinBold17&=&10000004
CONST KFontS60LatinBold19&=&10000005
CONST KFontS60NumberPlain5&=&10000006
CONST KFontS60ClockBold30&=&10000007
CONST KFontS60LatinClock14&=&10000008
CONST KFontS60Custom&=&10000009
CONST KFontS60ApacPlain12&=&1000000c
CONST KFontS60ApacPlain16&=&1000000d
```

```
rem
rem The following font consts are for UIQ devices only
rem e.g. Sony Ericsson P800
rem

CONST KFontUiqSwissABeta&=&017B4B0D

rem
rem End of font info.
rem

rem
rem I/O ACCESS
rem

rem For IOOPEN
rem Mode category 1
CONST KIoOpenModeOpen%=$0000
CONST KIoOpenModeCreate%=$0001
CONST KIoOpenModeReplace%=$0002
CONST KIoOpenModeAppend%=$0003
CONST KIoOpenModeUnique%=$0004

rem Mode category 2
CONST KIoOpenFormatBinary%=$0000
CONST KIoOpenFormatText%=$0020

rem Mode category 3
CONST KIoOpenAccessUpdate%=$0100
CONST KIoOpenAccessRandom%=$0200
CONST KIoOpenAccessShare%=$0400

rem
rem APPLICATION CREATION
rem

rem For FLAGS
CONST KFlagsAppFileBased%=1
CONST KFlagsAppIsHidden%=2

rem Language code for CAPTION
CONST KMaxLangsSupported%=33
CONST KLangEnglish%=1
CONST KLangFrench%=2
CONST KLangGerman%=3
CONST KLangSpanish%=4
CONST KLangItalian%=5
CONST KLangSwedish%=6
CONST KLangDanish%=7
CONST KLangNorwegian%=8
CONST KLangFinnish%=9
CONST KLangAmerican%=10
CONST KLangSwissFrench%=11
CONST KLangSwissGerman%=12
CONST KLangPortuguese%=13
CONST KLangTurkish%=14
CONST KLangIcelandic%=15
CONST KLangRussian%=16
CONST KLangHungarian%=17
CONST KLangDutch%=18
```

```
CONST KLangBelgianFlemish%=19
CONST KLangAustralian%=20
CONST KLangBelgianFrench%=21
CONST KLangAustrian%=22
CONST KLangNewZealand%=23
CONST KLangInternationalFrench%=24
CONST KLangCzech%=25
CONST KLangSlovak%=26
CONST KLangPolish%=27
CONST KLangSolvenian%=28
CONST KLangTaiwanChinese%=29
CONST KLangHongKongChinese%=30
CONST KLangPRCChinest%=31
CONST KLangJapanese%=32
CONST KLangThai%=33

rem MIME priority values
CONST KDataTypePriorityUserSpecified%=KMaxInt%
CONST KDataTypePriorityHigh%=10000
CONST KDataTypePriorityNormal%=0
CONST KDataTypePriorityLow%=-10000
CONST KDataTypePriorityLastResort%=-20000
CONST KDataTypePriorityNotSupported%=KMinInt%

rem
rem END OF CONST.OPH
rem
```

Appendix 3

Symbian Developer Network

A look at the resources, tools, SDKs and support that is available to the Symbian developer online.

A3.1 Symbian OS Software Development Kits

SDKs are built based on a particular reference platform (sometimes known as a 'reference design') for Symbian OS. A reference platform provides a distinct UI and an associated set of system applications for such tasks as messaging, browsing, telephony, multimedia, and contact/calendar management. These applications typically make use of generic application engines provided by Symbian OS. Reference platforms intended to support the installation of third-party applications written in native C++ have to be supported by an SDK that defines this reference platform, or at least a particular version of it. Since Symbian OSv6.0, four such reference platforms have been introduced, resulting in four flavors of SDK that can be found at the websites listed here:

- **UIQ** (***www.symbian.com/developer***)
- **Nokia Series 90** (***www.forum.nokia.com***)
- **Nokia Series 60** (***www.forum.nokia.com***)
- **Nokia Series 80** (***www.forum.nokia.com***)

Prior to this, SDKs were targeted at specific devices, such as the Psion netPad. Symbian no longer supports these legacy SDKs, but they are still available from Psion Teklogix at ***www.psionteklogix.com***.

For the independent software developer, the most important thing to know in targeting a particular phone is its associated reference platform. Then you need to know the Symbian OS version the phone is based on. This knowledge defines to a large degree the target phone as a platform for independent software development. You can then decide which SDK you need to obtain. In most cases you will be able to target – with a single

version of your application – all phones based on the same reference platform and Symbian OS version working with this SDK. The Symbian OS System Definition papers give further details of possible differences between phones based on a single SDK:

- **Symbian OS System Definition**
  www.symbian.com/developer/techlib/papers/SymbOS_def/symbian_os_sysdef.pdf
- **Symbian OS System Definition (in detail, inc Symbian OS v8.0)**
  www.symbian.com/developer/techlib/papers/SymbOS_cat/SymbianOS_cat.html

A3.2 Getting a UID for your Application

A UID is a 32-bit number, which you get as you need from Symbian. Every UIKON application should have its own UID. This allows Symbian OS to distinguish files associated with that application from files associated with other applications. UIDs are also used in other circumstances, such as to identify streams within a store, and to identify one or more of an application's views.

Getting a UID is simple enough. Just send email to ***uid@symbiandevnet.com***, titled 'UID request', and requesting clearly how many UIDs you want – 10 is a reasonable first request. Assuming your email includes your name and return email address, that's all the information Symbian needs. Within 24 hours, you'll have your UIDs.

If you're impatient, or you want to do some experimentation before using real UIDs, you can allocate your own UIDs from a range that Symbian has reserved for this purpose: 0x01000000–0x0fffffff. However, you should never release any programs with UIDs in this range.

Don't build different Symbian OS applications with the same application UID – even the same test UID – on your emulator or Symbian OS machine. If you do, the system will only recognize one of them, and you won't be able to launch any of the others.

A3.3 Symbian OS Developer Tools

As well as the following tools offerings from Symbian DevNet partners, Symbian DevNet provides a number of free and open source tools:

www.symbian.com/developer/downloads/tools.html

AppForge
Develop Symbian Applications Using Visual Basic and AppForge. AppForge development software integrates directly into Microsoft Visual

Basic, enabling you to immediately begin writing multi-platform applications using the Visual Basic development language, debugging tools, and interface you already know.

www.appforge.com

Borland
Borland offers C++BuilderX Mobile Edition and JBuilder Mobile Edition as well as the more recent Borland Mobile Studio for developers that want to develop rapidly on Symbian OS using C++, Java or both. These multi-platform IDEs offer on target debugging, GUI RAD, and a unifying IDE for Symbian OS SDKs and compilers.

www.borland.com

Forum Nokia
In addition to a wide range of SDKs, Forum Nokia also offers various development tools to download, including the Nokia Developer Suite for J2ME, which plugs into Borland's JBuilder MobileSet or Sun's Sun One Studio integrated development environment.

www.forum.nokia.com

Metrowerks
Metrowerks offer the following products supporting Symbian OS development:

- CodeWarrior Development Tools for Symbian OS Professional Edition
- CodeWarrior Development Tools for Symbian OS Personal Edition
- CodeWarrior Wireless Developer Kits for Symbian OS

www.metrowerks.com

Sun Microsystems
Sun provides a range of tools for developing Java 2 Micro Edition applications, including the J2ME Wireless Toolkit and Sun One Studio Mobile Edition.

http://java.sun.com

Texas Instruments
Development Tools for the OMAP Platform Easy-to-use software development environments are available today for OMAP application developers, OMAP Media Engine developers, as well as device manufacturers.

Tool suites that include familiar third-party tools and TI's own industry leading eXpressDSP DSP tools are available, allowing developers to easily develop software across the entire family of OMAP processors.

http://focus.ti.com

Symbian DevNet Tools
Symbian DevNet offers the following tools as an unsupported resource to all developers:

- **Symbian OS SDK add-ons**
  www.symbian.com/developer/downloads/tools.html
- **Symbian OS v5 SDK patches and tools archive**
  www.symbian.com/developer/downloads/archive.html

A3.4 Support Forums

The Symbian DevNet offers two types of support forum:

- **Support newsgroups**
  www.symbian.com/developer/public/index.html
- **Support forum archive**
  www.symbian.com/developer/prof/index.html

Symbian DevNet partners also offer support for developers:

Sony Ericsson Developer World
As well as tools and SDKs, Sony Ericsson Developer World provides a range of services including newsletters and support packages for developers working with the latest Sony Ericsson products, such as the Symbian OS powered P900.

http://developer.sonyericsson.com

Forum Nokia
As well as tools and SDKs, Forum Nokia provides newsletters, the Knowledge Network, fee-based case-solving, a Knowledge Base of resolved support cases, discussion archives, and a wide range of C++ and Java-based technical papers of relevance to developers targeting Symbian OS.

forum.nokia.com/main.html

Sun Microsystems Developer Services
In addition to providing a range of tools and SDKs, Sun also provides a wide variety of developer support services including free forums, newsletters, and a choice of fee-based support programs.

- **Forums**
  http://forum.java.sun.com

- **Support and newsletters**
  http://developer.java.sun.com/subscription

A3.5 Symbian OS Developer Training

Symbian's Technical Training team and Training Partners offer public and on-site developer courses around the globe.

- **Course dates and availability**
  www.symbian.com/developer/training

Early bird discount: Symbian normally offers a 20% discount on all bookings confirmed up to 1 month before the start of any course. This discount cannot be used in conjunction with any other discounts.

Course	Level	Language
Symbian OS essentials	Introductory	C++
Java on Symbian OS	Introductory	Java
Symbian OS: Application engine development	Intermediate	C++
Symbian OS: Application UI development	Intermediate	C++
Symbian OS: Internals	Advanced	C++
Symbian OS: UI system creation	Advanced	C++

Please note
Intermediate and advanced courses require previous attendance of OS Essentials. UI system creation course also requires previous attendance of Application UI course.

A3.6 Developer Community Links

These community websites offer news, reviews, features and forums, and represent a rich alternative source of information that complements the Symbian Development Network and the development tools publishers. They are good places to keep abreast of new software, and of course to announce the latest releases of your own applications.

My-Symbian
My-Symbian is a Poland-based website dedicated to news and information about Symbian OS phones. This site presents descriptions of new software for Symbian OS classified by UI. It also features discussion forums and an online shop.

http://my-symbian.com

All About Symbian
All About Symbian is a UK-based website dedicated to news and information about Symbian OS phones. The site features news, reviews, software directories, and discussion forums. It has strong OPL coverage.

www.allaboutsymbian.com

SymbianOne
SymbianOne features news, in-depth articles, case studies, employment opportunities and event information all focused on Symbian OS. A weekly newsletter provides up-to-date coverage of developments affecting the Symbian OS ecosystem. This initiative is a joint venture with offices in Canada and New Zealand.

www.symbianone.com

NewLC
NewLC is a French-based collaborative website dedicated to Symbian OS C++ development. It aims to be initially valuable to developers just starting writing C++ apps for Symbian OS, and with time cover more advanced topics.

www.newlc.com

infoSync World
infoSync World is a Norway-based site providing features, news, reviews, comments, and a wealth of other content related to mobile information devices. It features a section dedicated to Symbian OS covering new phones, software, and services – mixed with strong opinions that infoSync is not afraid to share.

symbian.infosyncworld.com

Your Symbian
Your Symbian (YS) is a fortnightly magazine distributed exclusively by email. YS takes a lighthearted look at the Symbian OS world. Major news is covered in its editorial and it includes a software round-up. To sign up, browse the archives, or get in touch with the editorial team.

www.yoursymbian.com

TodoSymbian (Spanish)
TodoSymbian is a Spain-based website for everyone wanting to read in Spanish about Symbian OS. It provides news, reviews, software directories, discussion forums, tutorials, and a developers section.

www.todosymbian.com

A3.7 Symbian OS Books

Symbian OS C++ for Mobile Phones, Vol. 2
Richard Harrison *et al.*
John Wiley & Sons, ISBN 0470871083

Symbian OS C++ for Mobile Phones, Vol. 1
Richard Harrison *et al.*
John Wiley & Sons, ISBN 0470856114

Symbian OS Explained
Jo Stichbury
John Wiley & Sons, ISBN 0470021306

Programming Java 2 Micro Edition on Symbian OS
Martin de Jode *et al.*
John Wiley & Sons, ISBN 047092238

Wireless Java for Symbian Devices
Jonathan Allin *et al.*
John Wiley & Sons, ISBN 0471486841

Symbian OS Communications Programming
Mike Jipping
John Wiley & Sons, ISBN 0470844302

Programming for the Series 60 platform and Symbian OS
Digia, Inc.
John Wiley & Sons, ISBN 0470849487

Developing Series 60 Applications
Edwards, Barker
Addison Wesley, ISBN 032126875X

A3.8 Open Source Projects

Many open source projects are happening on Symbian OS. They are a rich source of partially or fully functional code, which should prove useful to learn about use of APIs you're not yet familiar with. Please also consider contributing to any project that you have an interest in.

Repository Websites

SymbianOS.org
http://symbianos.org

Community website dedicated to the development of open source programs for Symbian OS. Hosted projects include: Vim, Rijndael encryption algorithm, MakeSis package for Debian GNU/Linux, etc.

Symbian open source
http://www.symbianopensource.com/

Repository for Symbian OS open-source software development. It provides free services to developers who wish to create, or have created, open source projects.

Open source for EPOC32
http://www.edmund.roland.org/osfe.html

Website of Alfred Heggestad, where he maintains a list of open source projects for Symbian OS.

Appendix 4

Specifications of Symbian OS Phones

Notes on the UI, screen size, and other attributes of Symbian OS phones relevant to OPL programmers. Technical information can also be found at: ***www.symbian.com/phones***

Please note that this is a quick guide to Symbian OS phones, some of which are not yet commercially available. The information contained within this appendix was correct at time of going to press. For full, up-to-date information refer to the manufacturer's website.

BenQ P30

OS Version	Symbian OS v7.0
UI	UIQ 2.1
Built-in memory available	32 MB
Storage media	MMC and SD
Screen	208×320 pixels 65,536 colors TFT
Data input methods	Keypad Pointing device
Camera	640×480 resolution
Network Protocol(s)	GSM E900/1800/1900 HSCSD GPRS (Class 10, B)
Connectivity	Infrared Bluetooth USB
Browsing	WAP 2.0 xHTML (MP)

Motorola A920/A925

OS Version	Symbian OS v7.0
UI	UIQ 1.0
Built-in memory available	8 MB
Storage media	MMC and SD
Screen	208×320 pixels 65,536 colors TFT
Data input methods	Small number of keys Pointing device
Camera	640×480 resolution
Network Protocol(s)	GSM 900/1800/1900 HSCSD GPRS (Class 4) 3G
Connectivity	Infrared Bluetooth (A920 No/A925 Yes) USB Serial
Browsing	xHTML (MP)

Motorola A1000

OS Version	Symbian OS v7.0
UI	UIQ 2.1
Built-in memory available	24 MB
Storage media	Triflash-R
Screen	208×320 pixels 65,536 colors TFT
Data input methods	Small number of keys Pointing device
Camera	1280×960 resolution 4×digital zoom
Network Protocol(s)	GSM 900/1800/1900 WCDMA 2100 HSCSD GPRS (Class 10, B) EDGE 3G
Connectivity	Bluetooth USB
Browsing	WAP XHTML (MP)

Nokia 3230

OS Version	Symbian OS v7.0s
UI	Series 60 v2
Built-in memory available	6 MB
Storage media	RS-MMC
Screen	176×208 65,536 colors
Data input methods	Keypad
Camera	1.3 megapixel resolution 3×digital zoom
Network Protocol(s)	GSM 900/1800/1900 HSCSD GPRS (Class 10) EDGE
Connectivity	Bluetooth Infrared USB
Browsing	WAP 2.0 xHTML HTML

Nokia 3600/3650

OS Version		Symbian OS v6.1
UI		Series 60 v1
Built-in memory available		3.4 MB
Storage media		MMC
Screen		176×208 4096/65,536 colors
Data input methods		Keypad
Camera		640×480 resolution
Network Protocol(s)	*3600*	GSM 850/1900
	3650	GSM 900/1800/1900 HSCSD GPRS (Class 8; B)
Connectivity		Infrared Bluetooth
Browsing		WAP 1.2.1 xHTML

Nokia 3620/3660

OS Version		Symbian OS v6.1
UI		Series 60 v1
Built-in memory available		4 MB
Storage media		MMC
Screen		176×208 4096/65,536 colors
Data input methods		Keypad
Camera		640×480 resolution
Network Protocol(s)	*3620*	GSM 850/1900
	3660	GSM 900/1800/1900 HSCSD GPRS (Class 8; B)
Connectivity		Infrared Bluetooth
Browsing		WAP 1.2.1 XHTML (MP)

Nokia 6260

OS Version	Symbian OS v7.0s
UI	Series 60 v2
Built-in memory available	3.5 MB
Storage media	MMC
Screen	176×208 65,536 colors TFT
Data input methods	Keypad
Camera	640×480 resolution 4×digital zoom
Network Protocol(s)	GSM 900/1800/1900 GSM 850/1800/1900 HSCSD GPRS (Class 6, B)
Connectivity	Infrared Bluetooth USB
Browsing	HTML xHTML WAP 2.0

Nokia 6600

OS Version	Symbian OS v7.0s
UI	Series 60 v2
Built-in memory available	6 MB
Storage media	MMC
Screen	176×208 65,536 colors TFT
Data input methods	Keypad
Camera	640×480 resolution 2×digital zoom
Network Protocol(s)	GSM 900/1800/1900 HSCSD GPRS (Class 8; B and C)
Connectivity	Infrared Bluetooth
Browsing	WAP 2.0 xHTML (MP)

Nokia 6620

OS Version	Symbian OS v7.0s
UI	Series 60 v2
Built-in memory available	12 MB
Storage media	MMC
Screen	176×220 65,536 colors TFT
Data input methods	Keypad
Camera	640×480 resolution
Network Protocol(s)	GSM 850/1800/1900 GPRS (Class 8; B) HSCSD EDGE
Connectivity	Infrared Bluetooth USB
Browsing	WAP 2.0 XHTML (MP)

Nokia 6630

OS Version	Symbian OS v8.0
UI	Series 60 v2.6
Built-in memory available	3.5 MB
Storage media	MMC
Screen	176×208 pixels 65,536 colors TFT
Data input methods	Keypad
Camera	1280×960 resolution 6×digital zoom
Network Protocol(s)	GSM 900/1800/1900 WCDMA 2000 GPRS (Class 10, B) EDGE 3G
Connectivity	Bluetooth USB
Browsing	WAP 2.0 HTML xHTML

Nokia 6670

OS Version	Symbian OS v7.0s
UI	Series 60
Built-in memory available	8 MB
Storage media	RS-MMC
Screen	176×208 pixels 65,536 colors
Data input methods	Keypad
Camera	1152×864 resolution 4×digital zoom
Network Protocol(s)	GSM 850/900/1800/1900 GPRS (Class 6, B) HSCSD
Connectivity	Bluetooth USB
Browsing	WAP 2.0 HTML xHTML

Nokia 7610

OS Version	Symbian OS v7.0s
UI/Category	Series 60 v2.1
Built-in memory available	8 MB
Storage media	RS-MMC
Screen	176×208 pixels 65,536 colors TFT
Data input methods	Keypad
Camera	1152×864 resolution 4×digital zoom
Network Protocol(s)	GSM 850/900/1800/1900 GPRS (Class 10; B)
Connectivity	Bluetooth USB
Browsing	WAP 2.0 XHTML

Nokia 7650

OS Version	Symbian OS v6.1
UI	Series 60 v1
Built-in memory available	4 MB
Storage media	MMC
Screen	176×208 pixels 4096 colors
Data input methods	Keypad
Camera	640×480 resolution
Network Protocol(s)	GSM 900/1800 HSCSD GPRS (Class 6; B and C)
Connectivity	Infrared Bluetooth
Browsing	WAP 1.2.1

Nokia 7710

OS Version	Symbian OS v7.0s
UI	Series 90
Built-in memory available	80 MB
Storage media	MMC
Screen	640×320 pixels 65,536 colors TFT
Data input methods	Keypad Pointing device
Camera	1152×864 resolution 2×digital zoom
Network Protocol(s)	GSM 900/1800/1900 HSCSD GPRS (Class 10) EDGE
Connectivity	Bluetooth USB
Browsing	HTML xHTML

Nokia 9210i

OS Version	Symbian OS v6.0
UI	Series 80
Built-in memory available	40 MB
Storage media	MMC
Screen	640×200 pixels 4096 colors
Data input methods	Keypad Keyboard Customizable buttons beside screen
Camera	No
Network Protocol(s)	GSM 900/1800 HSCSD
Connectivity	Infrared Serial
Browsing	WAP 1.1 xHTML (MP)

Nokia 9300

OS Version	Symbian OS v7.0s
UI	Series 80
Built-in memory available	80 MB
Storage media	MMC
Screen	Two displays, both 65,536 colors main screen: 200×640 pixels secondary screen: 128×128 pixels
Data input methods	Keypad Full keyboard Customizable buttons beside screen
Camera	No
Network Protocol(s)	GSM E900/800/1900 EDGE GPRS (Class 10, B) HSCSD
Connectivity	Infrared Bluetooth USB
Browsing	HTML 4.01 xHTML

Nokia 9500

OS Version	Symbian OS v7.0s
UI	Series 80
Built-in memory available	80 MB
Storage media	MMC
Screen	Two displays, both 65,536 colors main screen: 200×640 pixels secondary screen: 128×128 pixels
Data input methods	Keypad Full keyboard Customizable buttons beside screen
Camera	640×480 resolution
Network Protocol(s)	GSM 850/900/1800/1900 HSCSD GPRS (Class 10, B) EDGE WiFi
Connectivity	Infrared Bluetooth USB
Browsing	HTML 4.01 xHTML

Nokia N-Gage

OS Version	Symbian OS v6.1
UI	Series 60 v1
Built-in memory available	4 MB
Storage media	MMC
Screen	176×208 4096 colors
Data input methods	Keypad
Camera	No
Network Protocol(s)	GSM 900/1800/1900 HSCSD GPRS (Class 6, B and C)
Connectivity	Bluetooth USB
Browsing	WAP 1.2.1 xHTML (MP)

Nokia N-Gage QD

OS Version	Symbian OS v6.1
UI	Series 60 v1
Built-in memory available	3.4 MB
Storage media	MMC
Screen	176×208 pixels 4096 colors
Data input methods	Keypad
Camera	No
Network Protocol(s)	GSM 850/900/1800/1900 HSCSD GPRS (Class 6, B)
Connectivity	Bluetooth
Browsing	WAP 1.2.1 XHTML (MP)

Panasonic X700

OS Version	Symbian OS v7.0s
UI/Category	Series 60
Built-in memory available	4 MB
Storage media	miniSD
Screen	176×280 pixels 65,536 colors TFT
Data input methods	Keypad
Camera	640×480 resolution
Network Protocol(s)	GSM E900/1800/1900 GPRS (Class 10; B)
Connectivity	Infrared Bluetooth USB
Browsing	WAP 2.0 xHTML (MP)

Sendo X

OS Version	Symbian OS v6.1
UI	Series 60
Built-in memory available	12 MB
Storage media	MMC and SD
Screen	176×220 65,536 colors
Data input methods	Keypad
Camera	640×480 resolution
Network Protocol(s)	GSM 900/1800/1900 GPRS (Class 8; B)
Connectivity	Infrared Bluetooth USB Serial
Browsing	WAP 2.0 xHTML (MP)

Siemens SX1

OS Version	Symbian OS v6.1
UI	Series 60
Built-in memory available	3.5 MB
Storage media	MMC
Screen	176×220 65,536 colors TFT
Data input methods	Keypad
Camera	640×480 resolution
Network Protocol(s)	GSM 900/1800/1900 HSCSD GPRS (Class 10; B)
Connectivity	Infrared Bluetooth USB
Browsing	WAP 2.0 xHTML (MP)

Sony Ericsson P800

OS Version	Symbian OS v7.0
UI	UIQ 2.0
Built-in memory available	12 MB
Storage media	Sony MS Duo
Screen	208×320 pixels (Flip Open); 208×144 pixels (Flip Closed) 4096 colors TFT
Data input methods	Flip keypad Pointing device Jog dial
Camera	640×480 resolution
Network Protocol(s)	GSM 900/1800/1900 HSCSD GPRS (Class 8, B)
Connectivity	Infrared Bluetooth USB support
Browsing	WAP 2.0 xHTML (MP)

Sony Ericsson P900

OS Version	Symbian OS v7.0 (+ security updates and MIDP2.0)
UI	UIQ 2.1
Built-in memory available	16 MB
Storage media	Sony MS Duo
Screen	208×320 pixels (Flip Open); 208×208 pixels (Flip Closed) 65,536 colors TFT
Data input methods	Flip keypad Pointing device Jog dial
Camera	640×480 resolution
Network Protocol(s)	GSM 900/1800/1900 HSCSD GPRS (Class 10; B)
Connectivity	Infrared Bluetooth USB support
Browsing	WAP 2.0 xHTML (MP)

Sony Ericsson P910

OS Version		Symbian OS v7.0
UI		UIQ 2.1
Built-in memory available		64 MB
Storage media		Memory Stick Duo Pro
Screen		208×320 pixels 262,000 colors TFT
Data input methods		Flip keypad Thumb keyboard Pointing device Jog dial
Camera		1152×864 resolution 4× digital zoom
Network Protocol(s)	*P910i*	GSM 900/1800/1900
	P910c	GSM 900/1800/1900
	P910a	GSM 850/1800/1900
Connectivity		Bluetooth Infrared USB support
Browsing		WAP 2.0 cHTML

Index

Indexed by TERRY HALLIDAY, Indexing Specialists Ltd.