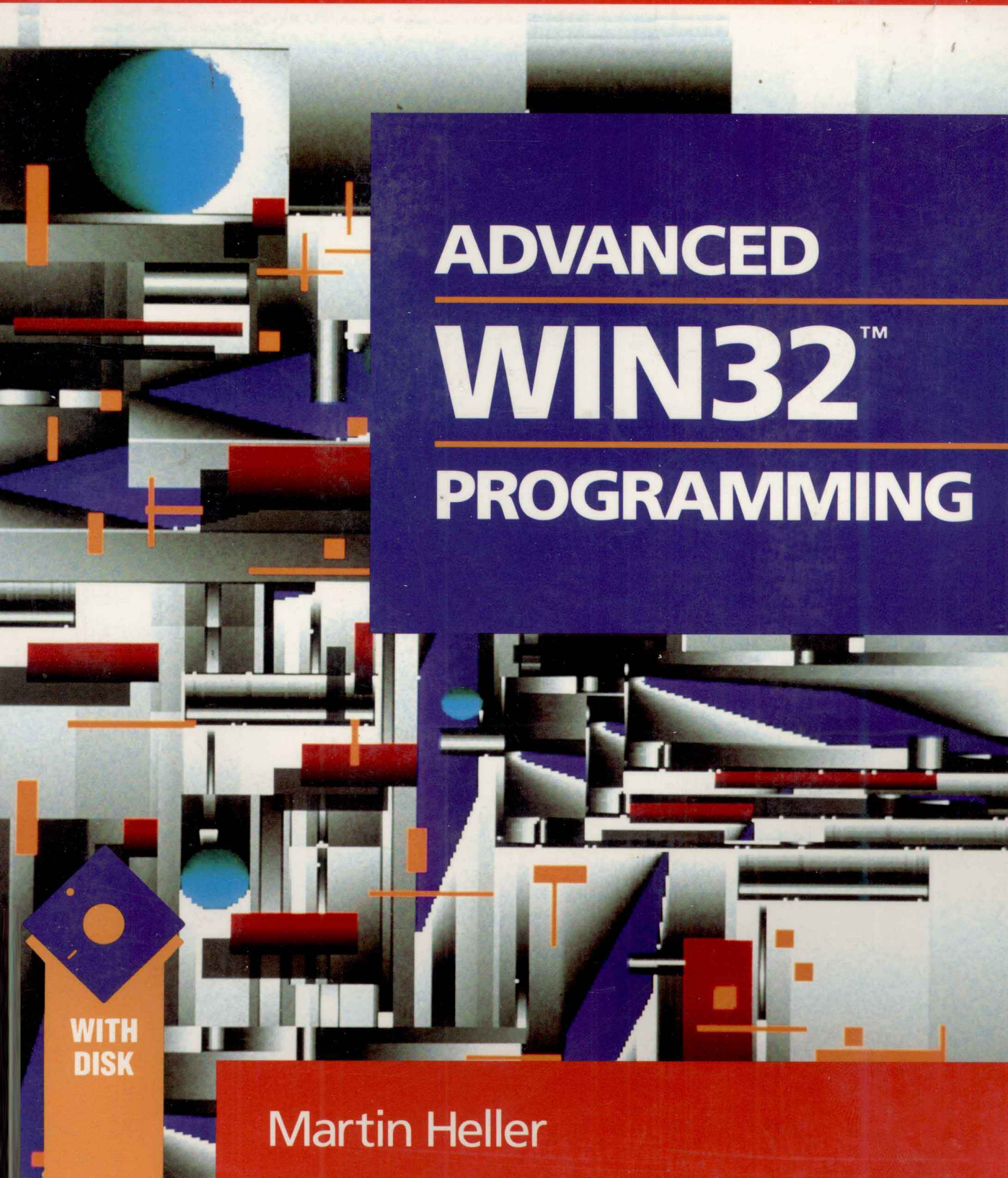
WILEY

WILEY PROFESSIONAL COMPUTING



ADVANCED WIN32™ PROGRAMMING

WITH DISK

Martin Heller

Advanced Win32™ Programming

Martin Heller

with Foreword

by

Paul Maritz, Microsoft Corporation

John Wiley & Sons, Inc.

New York Chichester Brisbane Toronto Singapore

For Eden

Library of Congress Cataloging-in-Publication Data

Heller, Martin
Advanced Win32 Programming / Martin Heller.
p. cm.
Includes index.
ISBN 0-471-59245-5 (book/disk set)
1. Application software. 2. Microsoft Win32 API. I. Title.
QA76.76.A65H45 1993
005.4'3—dc20 93-550
CIP

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

About the Author

Martin Heller develops software, writes, and consults in Andover, MA. You can contact him on BIX and MCI Mail as **mheller**, on CompuServe as **74000,2447**, and by mail care of John Wiley & Sons. Dr. Heller is the author of *Advanced Windows Programming*, also published by Wiley, as well as *Advanced Win32 Programming*.

Martin is a senior contributing editor and regular columnist for *WINDOWS* Magazine and the author of half a dozen PC software packages. He has been programming for Windows since early in the Windows 1.0 alpha test period. He has baccalaureate degrees in physics and music from Haverford College as well as Sc.M. and Ph.D. degrees in experimental high-energy physics from Brown University.

Dr. Heller has worked as an accelerator physicist, an energy systems analyst, a computer systems architect, a company division manager, and a consultant. Throughout his career he has used computers as a means to an end, much as a cabinet maker uses hand and power tools.

Martin wrote his first program for a drum-based computer in machine language in the early 1960s. (No, not assembly language, machine language.) The following year he taught himself Fortran II, and wrote mathematical programs in that language throughout high school.

In graduate school Martin wrote hundreds of programs in MACRO-9 assembler for a DEC PDP-9 computer, and hundreds more Fortran IV, APL, and PL/1 programs for an IBM 360/67. For his Ph.D. thesis he analyzed 500,000 frames of bubble chamber film taken at Argonne National Laboratory and helped take other data at Fermi National Accelerator Laboratory.

At New England Nuclear Corporation (currently a DuPont subsidiary) Dr. Heller developed an automatic computer data-acquisition and control system for an isotope-production cyclotron using Fortran IV+ and MACRO-11 on a PDP-11, with additional embedded 6802-based controllers. When the company acquired a VAX, Martin wrote one of the earliest smart terminal programs, in assembly language for the PDP-11 running RSX-11M.

At Physical Sciences Inc. (PSI) Martin developed a steady-state model of an experimental fuel cell power plant (under contract to the U.S. Department of Energy) in BASIC on a TRS-80 Model 3, and designed more advanced plants in BASIC on an early IBM PC. He developed a DOT-compliant crash sled data analysis program and a brake-testing data-acquisition, control, and analysis system in compiled BASIC for General Motors; he also developed the suite of programs that allowed General Motors to successfully defend itself against a government action over X-car braking systems.

Martin designed and developed MetalSelector, a materials selection and materials properties database program, under contract to the American Society for Metals (currently called ASM International), still in compiled BASIC. He designed EnPlot for the Society's graphing needs, intending the program for Windows 1.0, and put together a team of programmers to write it in C. When Windows 1.0 started slipping its schedule, Dr. Heller and his team implemented EnPlot for DOS instead of Windows.

Martin responded to the ongoing needs of the materials properties community by designing and implementing MetSel2 (at PSI) and later MatDB, in C for DOS, and EnPlot 2.0 for Windows (in both cases as an independent consultant). EnPlot is currently at revision 3.0 (and counting), and runs under Windows 3.0 and above.

While still at PSI, Martin designed two statistical subroutine libraries in Fortran for John Wiley & Sons. Statlib.tsf was a time-series and forecasting library, and Statlib.gl was a device-independent graphing library built on the GKS graphics standard. Both packages are now out of print.

As a consultant, Martin has worked for companies of all sizes to develop, design, improve, and/or debug Windows applications, and has performed strategic business consulting for large multinational corporations. His latest solely developed program is Room Planner, a meeting and conference layout system for the hospitality industry.

Foreword

It seems a software developer's work is never done. Develop an application for one desktop operating system and—you hope—users will want to run it on other systems, too. Port that application to another system and—if you're lucky—users will want to run it not just on the desktop, but also on the increasingly diverse range of microprocessor-based consumer electronics products and computers, including office equipment, handheld devices, portable computers, and very powerful server machines that are mainframes in all but name. Since this growing class of devices can address the computing needs of a widening set of applications, users will be looking to developers to supply not only personal productivity products, but also line-of-business or mission-critical applications.

How many different programming interfaces, operating systems, and form factors will developers have to address as computing options proliferate—and what will be the costs of doing so? The most useful approach for developers is one that lets them preserve and leverage their existing investments of time, money, and code as they address these broad choices. Given the very large base of investment in the industry in and around the Microsoft™ Windows™ operating system and Windows-based products, it makes sense for Windows to offer a cost-effective approach to development, helping to ensure that developers and users of the Windows operating system can easily gain access to new devices and address new applications by building on their existing investment and support structures.

But that's easier said than done. It is very difficult to address today's multiple development opportunities with a single underlying implementation of an operating system. The techniques and approaches one uses in a portable device, where low memory and power usage is paramount, are very different from the techniques and approaches that one needs in a high-end workstation or server, where security, capacity, and reliability are paramount. These challenges need to be overcome in order to gain the significant benefits of preserving a common user interface and common programming interfaces.

To address this challenge, Microsoft has been expanding Windows from its origin as a single product into a family of products where each family member has a specialized internal implementation but complements the others.

Our strategy for Windows is to develop a single coherent operating system family that is completely scalable. It spans computers from across pen devices, notebooks, desktop machines, high-end workstations, and even servers and multiprocessor machines. A single consistent operating system family with a common user interface, applications, and programming model brings real benefits to both software developers and their customers:

- Software developers gain a larger market and can save time and money that they might otherwise spend rewriting applications for each new platform and then supporting each platform-specific product. They can come to market faster and at lower cost, while minimizing their concern about investing in a platform with limited market acceptance. Vendors of all types can gain greater freedom to choose their business partners, because everyone speaks a common operating system language. They can also leverage their operating system knowledge to reduce costs.
- MIS managers, administrators, and other support personnel benefit because training, not hardware, is the largest cost in most computer system installations. A single operating system family can reduce training and administration costs for years to come; it simplifies and improves administration, leverages mini- and mainframe legacy systems, and eliminates the costs and headaches of changing over to new systems and platforms.
- End-users benefit, too. They can use an application on one platform (for example, their desktop), and know it will operate the same way on another platform (such as a pen-based sub-notebook). Once they have mastered one application, they will know how to work with completely different ones, because their various applications will work according to a consistent set of principles set by a single operating system. For end-users, this can save time and money; it can also boost productivity and effectiveness.

That is the strategy behind the expanding family of Windows operating system products. We are implementing this strategy through a scalable range of solutions, including:

- A Windows-compatible operating system for Microsoft At Work handhelds and small mobile devices
- Windows 3.1 for midrange and standalone PCs
- Windows™ for Workgroups for networked workgroup computing

- Windows NT, a 32-bit operating system for powerful PCs, workstations, and large organizational networks requiring a great client-server solution

For developers, the key to accessing these Windows solutions is the Win32 Application Programming Interface, a from-the-ground-up 32-bit programming interface for application development. Whether you're developing an entirely new application, or porting existing software to Windows from another system, such as UNIX or OS/2, Win32 allows you to write 32-bit applications that are compatible across the family of Windows solutions. Best of all, Win32 allows you to leverage the full advantage of Windows NT—the most powerful, reliable, and open platform for client-server computing. As corporate users increasingly turn to client-server computing for mission-critical and line-of-business solutions, developers will succeed based on their ability to harness Windows NT through Win32 application development.

How can you leverage Win32 to develop powerful, 32-bit applications for Windows NT and the Windows family? With *Advanced Win32 Programming*, you've already taken an important step. Martin Heller has done a masterful job of providing an all-in-one resource to Win32 development. In the pages that follow, you'll find a grand tour of Win32, including clear, abundant, and eminently practical step-by-step instructions and examples. Heller articulates the distinct issues you'll face, whether you're coming to Win32 from 16-bit development, porting from another 32-bit system, or creating an entirely new application.

Heller shows you how to make the transition from the C programming language to C++; how to optimize your Win32 application for Windows NT; how to run a Win32 application on Windows 3.1 using Win32s; even how to support multimedia and pen computing. With Heller at your side, you'll master networking, interprocess communications, and related mechanisms for distributed client-server solutions. You'll even discover how to use Unicode to create international versions of your application. With your Win32-based applications, you'll be able to fully leverage the power, reliability, and openness of Windows NT. Here's a closer look at these advantages:

- Windows NT delivers its power through the same Windows interface and technology already familiar to millions of Windows users. Win32 builds on that advantage by making advanced operating system capabilities available to applications through features such as multithreaded processes, synchronization, security, I/O, and object management. Your users can run more applications, and more powerful ones, at once. Because Windows NT is platform independent and scalable, users will be able to run your Win32 applications on processors ranging from Intel® X86 chips and RISC chips from MIPS and Digital

to some 30 symmetric multiprocessing systems—and on more than 2,600 computers and peripherals.

- Users need your 32-bit applications to run in the most reliable environment possible. Reliability and security have been "designed" into Windows NT, not added on as a layer afterwards. Features including uninterruptible power supply support and the new Windows NT file system (NTFS), minimize the chance of hardware failure and help ensure fast recovery from any exceptional failures that do occur. Windows NT provides comprehensive security and is government-certifiable at the C2 level of security to guard against inadvertent or malicious tampering. That enables Windows NT—and your Win32 applications—to penetrate new markets.

- Openness is another strategic criterion for a 32-bit operating system. The Windows NT operating system has built-in support for multiprotocol networking, including TCP/IP, NetBEUI, IPX/SPX, and DLC. It supports most networks, including SNA, LAN Manager, NetWare, NFS, Banyan® VINES, and AppleTalk. Windows NT supports distributed computing standards, including Windows Sockets, Named Pipes, and OSF DCE–compatible Remote Procedure Calls (RPC). This combined support makes Win32 an excellent choice for your distributed client-server applications—they can seamlessly access information on different hosts and databases throughout a network.

Understandably, industry participants share our enthusiasm for Windows NT. By the time Windows NT was introduced in mid-1993, more than 73,000 of them had already purchased the Windows NT software development kit, making it one of the best-selling operating systems software development kits ever. Independent software vendors have more than 2,000 32-bit applications for Windows NT under active development. Corporate users are developing another 3,800 applications for in-house use. More than 25 percent are native ports from UNIX®, VMS®, OS/400, and other high-end systems.

Welcome to the world of Win32 and Windows NT—and best wishes for success as you discover the benefits of 32-bit Windows application development for yourself.

Paul Maritz
Senior Vice President
Systems Division
Microsoft Corporation

Preface

Advanced Win32 Programming is, in a sense, a travel guide or road map. It maps the territory we software developers need to travel from the familiar 16-bit world of Windows to the 32-bit frontier, programming for Windows NT and Win32s.

Why do we have to leave our familiar haunts? The advantages of moving to 32-bit programming from 16-bit programming per se are compelling: many computations speed up automatically, the confining 64 KB segment size becomes a roomier 4 GB linear memory space, and most of the annoying overflow problems having to do with integer ranges disappear. In specific cases, the speedup is dramatic: for instance, when you turn computations that use "huge" 16-bit segment:offset pointers (like the 24-bit color adjustment implemented in the IMAGE2 example in my previous book, *Advanced Windows Programming*) into computations that use "near" 32-bit pointers, you get roughly a factor of five speedup.

Besides the inherent advantages of a 32-bit system, Windows NT has a number of new features that make it attractive. Windows NT supports high-end hardware like MIPS R4000 boxes, DEC Alpha AXP PCs, and symmetric multiprocessing machines. It has an enhanced graphical device interface that includes Bézier curves (a useful family of smooth curves similar to splines), paths (a generalized mechanism for creating and filling complex figures), world transforms (which allow the displayed graphics page to be rotated, scaled, reflected, and/or sheared from the drawing coordinate system), and masking (which allows regions to be excluded from display of a bitmap). It supports multiple threads of execution (a lightweight form of multitasking), several interprocess communication mechanisms, and C2-level security. And, not least, Windows NT comes with substantial networking support.

Win32, the API for Windows NT, is largely compatible with the 16-bit Windows 3.1 API. The basic differences: existing APIs have been widened to 32 bits; new APIs support threads, multitasking, security, Bézier curves, and other advanced features of Windows NT; and, finally, irrelevant 16-bit APIs, such as segment manipulation, have been dropped.

Win32s, a set of DLLs and VxDs that can help you increase the market for your Win32 programs by letting them run on Windows 3.1, is useful and convenient but not a panacea. The new APIs in the Win32 set that support advanced features of Windows NT appear in Win32s (the "s" stands for "subset") only as stubs that return FALSE. You can, for instance, write a Win32s program that tries to create a thread—but you must write the program to work even if the thread creation fails, as it will when running on Windows 3.1.

Win32s is not a completely general solution for building 32-bit Windows applications, either. One major limitation is that it has no provision for directly calling functions in 16-bit DLLs other than supported system functions. Win32 does, however, support several ways to call 16-bit DLL functions indirectly.

Future versions of Windows—starting with "Chicago"—will include much of the new functionality that is in Windows NT. A Win32s program, written to take advantage of advanced functions when they are present, will automatically work better and faster on newer versions of Windows than it does on Windows 3.1. If you want to prepare for "Chicago" learn Win32 now.

I decided to write *Advanced Win32 Programming* for selfish reasons: I wanted to force myself to use the new functionality in Windows NT, and to force myself to migrate from C programming to C++ programming. I've tried to keep a good enough log of my journey for it to be useful as a map for others attempting similar journeys.

I hope you'll find *Advanced Win32 Programming* useful as a guide, whether you're porting existing code from 16-bit to 32-bit Windows, writing new code compatible with both, or jumping into 32-bit Windows programming with both feet. Whether you're new to all this or an old hand, I hope you'll find some ideas and snippets of code here that you can use in your own work. If you haven't already done so I hope you'll also read my previous volume, *Advanced Windows Programming*, which takes you from familiarity with the Windows SDK to the point where you can write substantial, multimodule Windows programs.

To actually use the material in *Advanced Win32 Programming* you should have access to a computer running Windows NT, to copies of the Win32 documentation, to 32-bit C and C++ compilers for Windows NT, to a computer running Windows 3.1, and to copies of the Win32s libraries.

In Chapter 1, "Crossing the Great Divide," we learn to make the transition from 16-bit programming to 32-bit Windows programming. In Chapter 2, "Any Port in A Storm," we move the Image2 example developed in *Advanced Windows Programming* to Win32 as quickly as possible. Then in Chapter 3, "Upping the Ante," we learn to make the transition from C programming to the C++ programming language, encapsulating as we go.

We clean up our ported code, and think about reorganizing it as C++ with classes, in Chapter 4, "A Higher Standard." Then we learn to use some of the advanced features of Win32 in Chapter 5, "Total Immersion."

We take a step back and learn to trade off between the advanced features of Windows NT and compatibility with Windows 3.1, using Win32s, in Chapter 6, "With a Shoehorn." In Chapter 7, "A Little Song, A Little Dance," we begin to apply some multimedia programming; in Chapter 8, "The Pen Is Mightier," we learn to support Pens, Ink, and Tablets.

We learn to use Unicode and do internationalization in Chapter 9, "Lingua Franca," and in Chapter 10, "OLÉ Again," we investigate OLE 2 and perhaps even OLE 3. In Chapter 11, "Citizen of the Galaxy," we examine interprocess communication and distributed computing, and finally in Chapter 12, "The Far Horizon," we look into the future. In addition, for those of you looking for tools, we describe some resources and tools for Win32 Development in the Appendix, "Cornucopia."

That's a lot of material. Never fear: we'll start slowly, and there will be plenty of examples.

It's next to impossible to write a book of this kind without help. First and foremost, I'd like to thank my editor, Diane Cerra, for pushing—and pushing, and pushing—me to actually write the book. Like all writers, I prefer having written. It's okay, Diane: you can put away the bullet with my name on it until the next book.

I'd also like to thank my reviewers: Ed Adams, Roger Grossman, Timothy Larson, Chris Marriott, John Ruley, William vanRyper, and Bjarne Stroustrup. These gentlemen spent a lot of time telling me I was all wet and making me do better, and I appreciate their efforts—now that I have rewritten, and rewritten, and rewritten.

The clean design and layout of this book is the work of Claire Stone Spellman of Desktop Studios, aided and abetted by Frank Grazioli and his production team at Wiley. I did my writing in Microsoft Word for Windows and saved the figures as TIFF files; Claire took the files on diskettes and laid them out in PageMaker for the Macintosh. We'll get the bugs worked out of that process one of these days; meanwhile, Claire worked around a few of them with simple but time-consuming methods like recreating sections from scratch.

Finally, I'd like to thank my wife Claudia and my daughters Tirzah and Moriah for putting up with me as I muddled through another book. It's not easy living with a writer, and my family made sure I didn't ever forget it. Claudia in particular had plenty to put up with, but she won the race: our third child was born after I finished writing the body of the book, but before this book came off the press. It is to that child that I dedicate this book.

Contents

CHAPTER

1

In which we learn to make the transition from 16-bit programming to 32-bit Windows programming.

Crossing the Great Divide

There are many ways to get from the east coast of the United States to the west coast. Today, most people fly, but some drive, using one of several possible interstate highways. A hundred fifty years ago, small parties of hardy (or foolhardy) pioneers set out in horse-drawn wagons. They hoped to cross the deserts as quickly as possible, hoped to cross the mountains before the winter snows, hoped to avoid hostile natives.

Not everyone survived the trip. Without good directions, you could get lost in the desert without water, miss the mountain passes. A party whose wealthiest family was named Donner found out the value of good directions the hard way.

Follow me: I've been there and back. I'll get you through safely.

This first chapter is a sort of hazard map: it doesn't show you the best route, but it shows you where the mountains and deserts lie. Later on, we'll find out how to avoid most of the hazards.

Our first obstacle is the very change in word size that induced us to make the trip in the first place. It's a small obstacle; some might say even a trivial obstacle. But it has tripped up many a pioneer.

Word Size Annoyances

The influence of word size on programming is often both over- and underestimated. It is overestimated by people who believe that a larger word size *automatically* makes programs run faster and more accurately, and underestimated by people who ignore the capacity, range, and exception-handling issues that are important when the word size is too small for the problem at hand.

A 16-bit signed integer has a range of -32,768 to 32,767 (32K-1); a 16-bit unsigned integer has a range of 0 to 65535 (64K-1). A 32-bit signed integer has a range of -2,147,483,648 to 2,147,483,647 (2G-1) and a 32-bit unsigned integer has a range of 0 to 4,294,967,295 (4G-1).

Limits of 32K and 64K arise frequently in 16-bit Windows programming, sometimes directly because of the size of an integer, and sometimes indirectly because of the maximum size of a memory segment. Sometimes, the limit is a minor annoyance that can be avoided with careful coding; other times, the limit causes a major slowdown in the program. Some examples are in order.

First, a minor annoyance. You might want to find the centroid of a cluster of points in your display space. If the display space is 640 by 480 (the VGA screen) the following code will work correctly for 16-bit integers:

```
#define n 10
int x[n],y[n],xmid=0,ymid=0;

for(i=0;i<n;i++) {
      xmid += x[i];
      ymid += y[i];
      }
xmid /= n;
ymid /= n;
```

The code continues to work on a 1024 by 768 Super-VGA screen. But what happens when you go to print on a 300-dot-per-inch laser printer? The 8.5-inch by 11-inch page would have dot coordinates of 2,550 by 3,300. Once in a while—not often—ten coordinates will add up to more than 32,767, the sum will overflow, and you'll wind up drawing the centroid at the wrong end of the page (or even off the page) since the result will be a large *negative* number. Obviously, if **n** were bigger than 10 the error would occur more frequently.

In the case of the centroid, we can rewrite the code to be more reliable, albeit less efficient and less accurate, without resorting to **long** variables:

```
#define n 10
int x[n],y[n],xmid=0,ymid=0;

for(i=0;i<n;i++) {
      xmid += (x[i] / n);
      ymid += (y[i] / n);
      }
```

Another alternative for this particular problem would be to make **xmid** and **ymid** into **long** variables—that is, use 32 bits just for the working variables. In a 16-bit architecture, that means that the compiler has to generate code to extend the value of **x[i]** and **y[i]** from 16 bits to 32 bits, and has to generate or call code to add two 32-bit numbers using 16-bit registers. That's a poor substitute for the **add** instructions generated by the addition code in the original loop.[1]

What about a more serious case? Consider this code from the PROCESS module of IMAGE2, the final example program in *Advanced Windows Programming*:

```
int dr,dg,db;
double r,g,b,dh,ds,dv,dl,h,s,v,l;
LPBITMAPINFOHEADER lpbi;
RGBQUAD FAR *pRgb;
unsigned char huge *pixels;
unsigned char huge *pb;
unsigned char huge *pb1;
unsigned char huge *pb2;
int colors,i;
DWORD il;
//NOTE: BOUND is a macro that limits the first variable
//  to the range of the second and third variables

lpbi=(VOID far *)GlobalLock(hdibCurrent);
pixels = (unsigned char huge *)(lpbi + lpbi->biSize);
switch(ColorModel) {
      case IDD_RGB:
        for(il=0;il<lpbi->biSizeImage;il+=3) {
             pb=pixels+il;
             pb1=pixels+il+1;
             pb2=pixels+il+2;
             *pb  = (BYTE)BOUND(*pb +db,0,255);
             *pb1 = (BYTE)BOUND(*pb1+dg,0,255);
             *pb2 = (BYTE)BOUND(*pb2+dr,0,255);
             }
        break;
...
```

There are a couple of things to notice in this particular example, which is supposed to alter the colors in a 24-bit image. First of all, the logic is incorrect and the routine fails for some images: instead of looping over the pixels in the entire image linearly, the code should loop over pixels within scan-lines. Why? Because the ends of the scan-lines can be padded out so that the next line will fall on a double word boundary, which throws off the counting by threes. That would be bad enough, but there's worse to come: Windows doesn't necessarily even have memory assigned to the null bytes at the ends of scan-lines. If you found and fixed this bug youself,

[1] Consider the code generated by **c = a*b**, where all the variables are declared **long**. The 16-bit Microsoft compiler will generate a call to a subroutine that does roughly 5 multiplications and 2 additions. Almost any 32-bit compiler will generate inline code for this line that does one multiplication. (Thanks to Chris Marriott for pointing this out.)