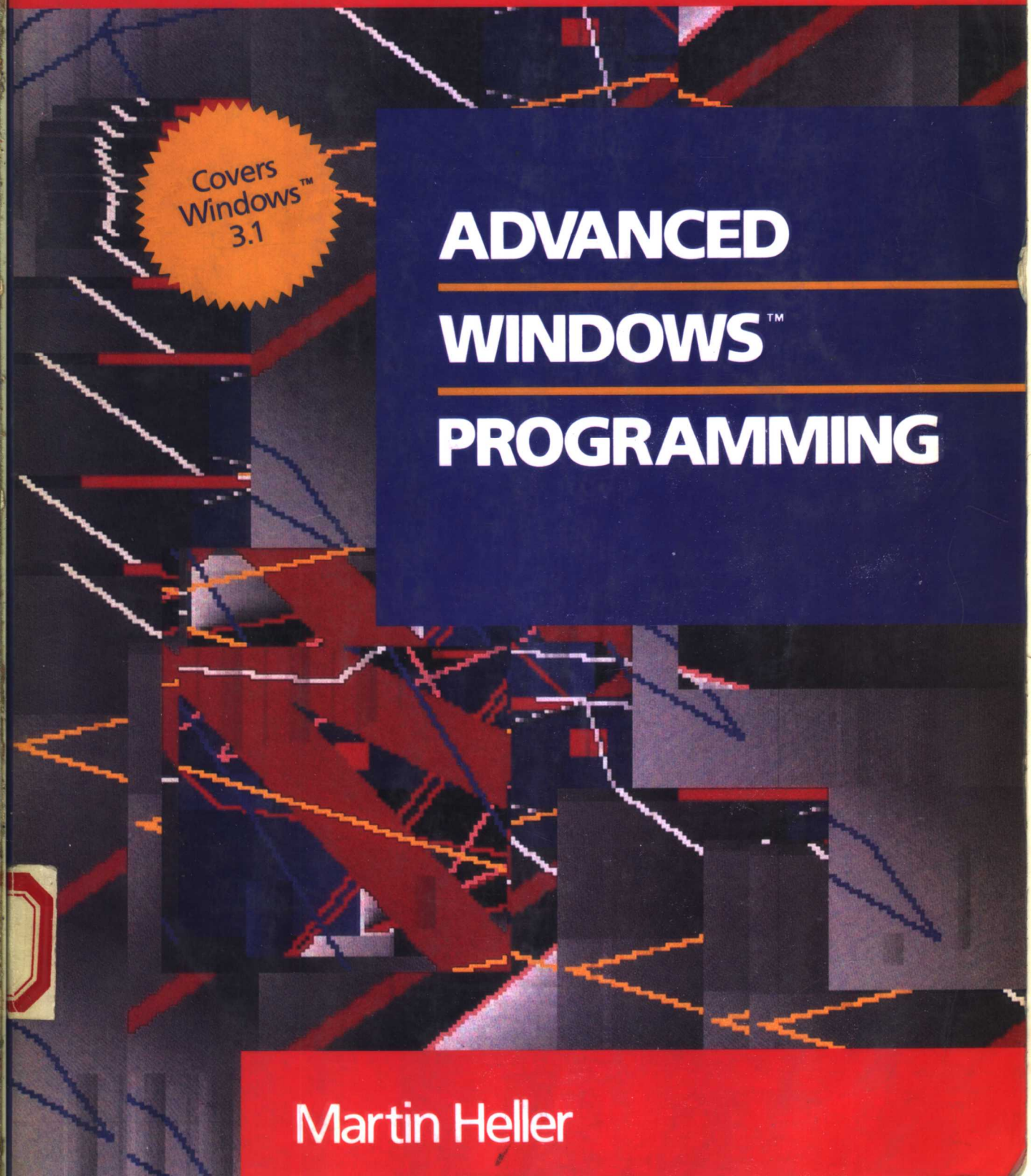
WILEY

WILEY PROFESSIONAL COMPUTING



ADVANCED WINDOWS™ PROGRAMMING

Martin Heller

Advanced Windows™ Programming

Martin Heller

John Wiley & Sons, Inc.

New York Chichester Brisbane Toronto Singapore

In recognition of the importance of preserving what has been written, it is a policy of John Wiley & Sons, Inc., to have books of enduring value published in the United States printed on acid-free paper, and we exert our best efforts to that end.

Library of Congress Cataloging-in-Publication Data

Heller, Martin, 1951—
Advanced Windows Programming / Martin Heller.
p. cm.
Includes bibliographical references and index.
ISBN 0-471-54711-5
ISBN 0-471-55172-4 (book/disk set)
1. Windows (Computer programs) 2. Microsoft Windows (Computer program) I. Title
QA76.76.W56H45 1992
005.4'3dc20 91-35881
CIP

Printed in the United States of America
10 9 8 7 6 5 4 3 2

Printed and bound by Malloy Lithographing, Inc..

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.

Martin is a 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 mate-

rials 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 with several companies to develop, design, and/or debug Windows applications. His latest solely developed program is **`Room Planner`**, a meeting and conference layout system for the hospitality industry.

Preface

Advanced Windows Programming was written to show the Windows developer the essentials of writing *real* Windows programs. Up until now the only ways to find out this information have been to program Windows for a couple of years, to look over the shoulders of experienced Windows developers, and to ask questions in a computer conference; the best such have been BIX (the Byte Information Exchange), CompuServe and GEnie. The material hasn't even been covered in course work, except at Microsoft University and one specialized school in—of all places—Iceland.

While the title says Advanced, this book will be of equal benefit to beginning Windows programmers. A good set of prerequisites would be:

- A strong working knowledge of the C programming language, with a firm understanding of pointers and structures.
- A good grasp of Intel segmented memory architecture.
- A more than passing acquaintance with Microsoft C, including the **`near`** and **`far`** keyword extensions.
- Some familiarity with the Microsoft Windows application program interface and message-passing model. This can be achieved by reading through the Microsoft Windows SDK (Software Development Kit) *Guide to Programming* manual, and Charles Petzold's *Programming Windows, Second Edition* (Microsoft Press, 1990).

To compile and run the examples in the book or companion disks as presented, you'll need Microsoft C V6.0 or later, the Microsoft Windows SDK V3.0 or later, and Microsoft Windows V3.0 or later. If you don't have Microsoft C, you can still use the programs from this book: Appendix 2 explains how to adapt the code for Zortech C++, Appendix 3 explains how to do the same for Borland C++, Appendix 4 covers the use of Watcom C, and Appendix 5 covers Quick C for Windows. You'll need a PC capable of running all of this, of course; a minimum configuration would be an 80286-based PC with 640KB of memory, a hard disk, an EGA display, and a mouse.

Note that the companion disks have contents that don't appear in the book, and vice versa. This is strictly a matter of space and practicality. Some material is best presented as it is here, in print; other material is best given in executable form. If you bought the book without the diskettes, you can order them using the attached card. Some, but not all of the contents of the companion disks are also available on BIX for downloading.

Once you've got the prerequisites under your belt—C and a little exposure to Windows at the introductory level—*Advanced Windows Programming* can help you get over the rather formidable mental barrier to writing Windows programs. Theres a *big* difference between writing a 500-line program in small model that does nothing but draw monochrome pictures on the screen, and a 5,000-line multiple-module program in mixed model (medium model plus global allocations) that deals with color, palette management, printers, fonts, the clipboard, and dynamic data exchange. Don't feel that you have to know every little thing about Windows before you start: the value of this kind of book is that you'll pick it up and gain fluency as you go, as long as you have enough background to follow the exposition.

In *Advanced Windows Programming* I cover the process of developing *real* Windows programs, as opposed to toy examples. After some introductory matter, the book revolves around a single major application to display device-independent bitmaps, and a number of smaller test fixtures to help us develop our modules.

We start, innocuously enough, with a look at the SHOWDIB example program from the Microsoft Windows Software Development Kit (SDK). We add capabilities: more file formats, some image processing algorithms.

Then we add editing capabilities using the Windows clipboard, demonstrating useful functions like dragging regions, handling global memory larger than 64K, and implementing undo functions. We continue by investigating DDE and OLE client, server, and remote execution capabilities. Along the way, we deal with the problems unique to large-sized Windows programs, such as memory management and modularization, and problems that occur only when two Windows programs interact.

We take an intermission from our bitmap display program—which by now has grown into something of an image-processing program—and discuss different ways to debug Windows programs. We'll emphasize techniques for solving actual problems, in some detail, and we'll deal with the lovely problems that come up when the debugger makes the bug go away, or the debugger kills the program before the bug we're looking for can express itself.

We'll digress further, and talk about edit controls. We'll demonstrate how to customize edit controls by responding to various messages in a dialog box, and how to customize further by subclassing. When we've taken this as far as we think reasonable, we'll reimplement the edit control ourselves and enhance it to handle multiple fonts, point sizes, and text attributes.

Finally, we'll add our custom edit control to the bitmap display program and use it for adding titles to an image. A little tuning, and we—author and reader—will have built a fairly interesting application that is useful in and of itself, and open to further enhancements by the reader. Just as icing on the cake, we'll go on to talk about how

we'd port this application—or less ambitious Windows applications—to Presentation Manager, X-Windows, DOS, and the Macintosh.

I couldn't have written this book in a vacuum. I am deeply indebted to the many Windows designers, developers, technical writers, and support people at Microsoft; to Charles Petzold, whose fine book provides a solid foundation on which mine builds; to Diane Cerra, Terri Hudson, and Katherine Schowalter at Wiley, who prodded me to turn a vague idea into a substantial volume; and to Claire Stone, who designed the book's layout and helped me over the hurdles of desktop typesetting. I thank my beta readers: John Butler, Michael Geary, Lee Hasiuk, Kyle Marsh, Jean-Marc Matteini, Barry Nance, Bill Neuenschwander, Dan Rubin, John Skiver, and Carl Sturmer, as well as Wiley's two anonymous reviewers.

I also thank the many programmers on BIX who contributed examples, made suggestions, or helped to test and debug my software, including Marc Adler, Mike Geary, David Jones, Barry Nance, Dan Rubin, Bill van Ryper, Jay Slupesky, Anders Thun, and Bert Tyler. I thank Dana Hudes for contributing some fine photographs, and Steve Rogers of Kodak for scanning the images. And finally I thank my wife, Claudia—because.

Contents

CHAPTER

1

In which we present a general introduction to the more advanced Windows programming concepts, such as interprocess communication, memory management, subclassing, and superclassing.

Introduction

In the beginning, C programmers write (or copy) **hello.c**, for which the specification is to print the words "hello, world". The ordinary C version of this, which works perfectly well in Unix, DOS, OS/2, and any number of other text-based operating systems, is simply:

```
#include <stdio.h>
main()
{
        printf("hello,world\n");
}
```

Kernighan and Ritchie tell us that getting this program entered, compiled, and run is the "big hurdle" to getting started as a C programmer; "everything else is comparatively easy."

The equivalent "hello" program for Microsoft Windows (found in Chapter 1 of Charles Petzold's fine introductory book, *Programming Windows*) amounts to three pages of text—some 80 lines of code—in three files: **hellowin.c** (the C language source code), **hellowin.def** (the module definition file), and **hellowin.mak** (the "make" file). Getting "hello" for Windows running is another big hurdle—but not so big a hurdle as understanding everything that goes on to make it run. An even bigger hurdle for beginning Windows

programmers is to venture beyond the manuals; the barrier here is as formidable as was the dreaded map designation *terra incognita* to fifteenth-century sailors.

Consider this book your Baedeker, your guide to unknown lands, written by one who has explored their blackest depths and returned to tell the tale.

Real-World Programs versus Toys

Nobody ever said **hello.c** was useful—other than as a learning tool. Neither is the "Hello, Windows" program. But there is more to a Windows program than there is to **hello.c**; Petzold's **hellowin**, the Windows SDK **generic** program and other readily available small examples do valuable service as templates. You'll never have to reinvent the obligatory parts of a Windows program: just copy an example, change the names, and start adding functionality.

Still, the programs you'll find in the Windows SDK and in other Windows programming books are mostly short, easily digestible examples—which also makes them toy programs. They sometimes do useful things, such as enumerate your fonts or take a window snapshot, but they don't generally push any of the size limits on Windows programs.

On the other hand, real-world Windows programs often do push the limits. What do I mean by "real-world"? Why, something genuinely useful. Word for Windows, the word processor I'm using to write this book, is a good example. Its executable, **winword.exe**, is almost 900KB of code and resources. Its dynamic link libraries add up to another 900KB. It has 500KB of data files, 350KB of document templates, 250KB of import and export filters, and a 400KB help file. It isn't big just to be big—it is big because it does a lot of different things.

There is an ethic among some programmers and Sixties counterculture types: *small is beautiful.* We must emend that specifically for computer software: given two programs that do the same thing at the same speed, we prefer the one that uses the fewest system resources. Small programs that don't do much or take a long time to run don't qualify as beautiful. But small programs that do one or two things elegantly may qualify as *good hacks.* There's a satisfaction in writing a good hack that you don't often get from writing a big potpourri of a program; but there are techniques needed for big programs you just won't find in little ones.

Back to Word for Windows. It's got several megabytes of code. Normally, the sum of a program's static data, stack, and local heap can't exceed 64KB. Can you imagine that Word for Windows would be possible if there weren't a way around that restriction?

A big program can't blithely assume that any global memory blocks it needs will be allocated successfully—the sheer size of the program almost guarantees

that it will have to deal with low memory situations. Word for Windows can't require that every font available for your printer also be available for your screen: if it did, Microsoft would be swamped with returns of the product from people without enough disk space to build all those screen fonts. It can't assume that every file it reads will be in the format it expects to read, or it would crash every time it did an import. It can't arbitrarily limit itself to a "reasonable" number of fonts—the dozen fonts that might be reasonable at a secretary's workstation would hardly make a dent in the hundreds of fonts needed by a desktop publishing shop.

It has to deal, instead, with the real world—or as real a world as exists on personal computers. And the ways Windows programs deal with these real-world issues and problems is the major subject of this book.

Other Windows books concentrate on short programs because they're easier to understand than long programs. And, perhaps, because they're easier to write than long programs. We don't have that luxury: to show the problems of size and their solutions, we need to work on a big program. But for comprehensibility, we'll explain fragments of code in small chunks; and for reliability, we'll test them in small chunks with little test fixtures. Not all of our test programs will necessarily be Windows programs: when possible, we'll test our functions in a more controlled environment (plain DOS or OS/2) before bringing them into the multitasking mayhem of Windows.

I learned some of the techniques covered in this book the hard way: I worked them out for myself in the course of developing Windows programs. Other techniques were supplied by Microsoft, and still others were shared with me by other Windows developers. I'll do my best to give credit where credit is due—but I hope that the originators of some of the tricks that I present will forgive me, since their names have been lost as their hacks have been passed orally from programmer to programmer.

A small admission: while I advocate carefully testing functions and program fragments outside of the program they will eventually be part of, I don't always practice what I preach. Lots of times, bending to time constraints, I'll write the code once in its final resting place, quickly check that it works as expected, and go on the next pressing matter. I usually get away with such shortcuts; but, when they backfire, the time spent to isolate and repair the problem can be costly. You'll have to decide for yourself how much insurance (in the form of bench testing) you want to buy, and how much risk you can afford to take.

Memory Management Issues

Microsoft Windows isn't really a single environment. Depending on how you want to count, it is either three or five different environments in version 3.0. That

count goes down to two different memory environments in Windows 3.1, at least under DOS. (The NT Windows environment is so different that we won't try to cover it here.)

Windows Memory Modes

To begin with, Windows 3.0 can run in real, standard, and enhanced modes. Windows 3.1 drops support for real mode, and runs only in standard and enhanced modes.

Real mode in Windows 3.0 is similar to Windows 2, and can use three memory modes itself: no EMS, small-frame EMS, and large-frame EMS. EMS stands for Expanded Memory Specification; it is also called LIM (for Lotus-Intel-Microsoft, the companies who specified the standard) or banked memory. You might want to check to see what memory mode your copy of Windows is using: to do so, pull down the **About** box from the Windows Program Manager's **Help** menu (Figures 1.1 and 1.2).

With no EMS in real mode—the *basic* memory mode—Windows has very little memory available for applications; it is dead certain that you won't be able to run two big applications simultaneously, and that an application big enough to be useful will constantly be discarding and loading code segments. If you want to stress-test your application to see if you've done your segmentation well, use this mode. You can force real mode without EMS by starting Windows 3.0 with the command line:

```
WIN /R /N
```

On a machine with more than 640KB of memory, **HIMEM.SYS** (which comes with Windows) makes an extra 64KB of memory available to Windows

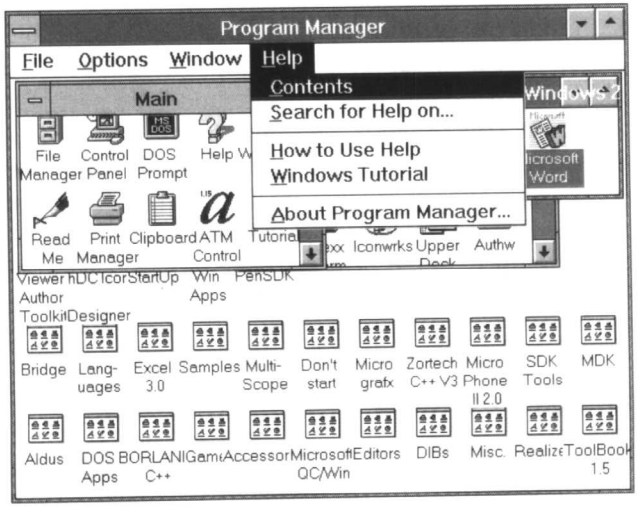


Figure 1.1. Pulling down the Program Manager Help Menu

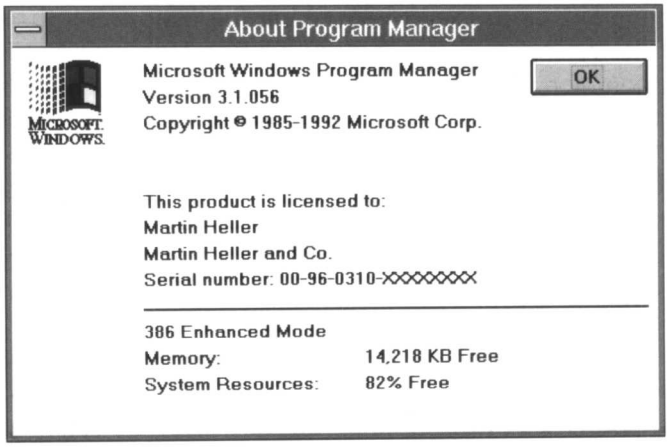


Figure 1.2. Memory display from Program Manager

through the use of the infamous A20 memory-addressing control line. (A20 refers to the pinout designation on the chip.) Basically, enabling A20 lets Windows use a quirk in the 80286 addressing scheme to reach the "High Memory Area," which resides just above the 1MB mark. Because of the way PCs are designed, only 640KB of RAM is mapped below the video adapters and ROM areas; additional memory is mapped above the 1MB line. To *really* stress-test your application for low memory situations, disable **HIMEM.SYS** temporarily and force Windows into basic mode—you're just about guaranteed to see segment motion, segment discarding, and segment loading. **HIMEM.SYS** is normally installed by the Windows 3 setup program; to disable it, edit your **CONFIG.SYS** file and reboot.

Small-frame EMS mode improves things somewhat for a single application: much of its code and data can be *banked*, or put into *expanded* memory. To get expanded memory in an 80286-based machine, you need a special expanded memory board (such as an Intel AboveBoard) that has bank-switching hardware. On an 80386- or 80486-based machine, you can emulate expanded memory using **EMM386.SYS**, which comes with Windows. You'll find instructions for installing **EMM386** in your Windows manual. You can also emulate expanded memory with third-party memory managers like 386-to-the-Max and QEMM.

You'll need to test your application in both EMS modes to ensure that it works correctly with bank-switching. To force small-frame EMS mode, start Windows 3.0 with the line:

```
WIN /R /E:999
```

Large-frame EMS mode gives each application its own banked memory, but leaves less low memory for each application than small-frame mode. The mix of low and banked memory is adjustable: vary the **/E:xxx** parameter on the

Windows command line to see the effect. Running Windows with the **/R** switch and without the **/E** switch will generally get Windows into large-frame EMS mode. Many commercial Windows applications will either fail to start or crash during operation in this mode, since most dynamic link libraries can't be banked and the low memory gets used up quickly. Unless you've marked your application for protected mode only (using the **/t** switch to RC) you need to test your own applications in large-frame mode to find out where the bank line needs to be set: you'll undoubtedly hear from customers who can't run your application because they're running Windows in large-frame EMS mode, and you'll need to know what to tell them.

Windows 3.0 standard mode uses the protected mode of the 80286 in much the same manner as does OS/2 1.x. Segment motion in this mode is handled by updating the segment descriptor table; the CPU uses the segment descriptor table to map segment handles to actual memory addresses. Handling the memory mapping in hardware improves the speed of Windows; even more speed improvement comes from the larger available address space. All the memory in standard mode is the same kind—you don't have any such thing as banked memory to worry about. The net result is that programs running in standard mode do a lot less discarding of segments than programs running in any of the real modes; the reduced disk activity and the reduced segment motion overhead combine to make standard mode a faster environment for Windows programs.

Standard mode allows one DOS session to run as a full-screen foreground task, but suspends the DOS session when it is in the background. This is fine on a 286, but doesn't use the full capabilities of a 386. Windows 3 enhanced mode, on the other hand, uses the advanced memory mapping modes of the 80386 to provide demand paged virtual memory, multiple DOS sessions, and some multitasking of sessions. If you've got a 386 box and enough memory ("enough" being theoretically more than 2 MB, or realistically at least 4 MB of RAM), enhanced mode will allow you to use disk as additional "RAM," run several DOS sessions at the same time you are running several Windows programs—and even run windowed DOS sessions. While the multitasking offered by Windows 3.0 in enhanced mode isn't as good as the multitasking offered by OS/2, it is good enough to let you run compiles while you edit, and play solitaire while you download.

Memory Use Guidelines

Every real application must be prepared to run out of memory and other system resources. Anytime you allocate or lock a memory block, you must check to see if the operation succeeded, and bail out cleanly if it hasn't. Unfortunately, this will clutter your code—but that's the price you pay for having a robust program. Paranoia in the proper places is a virtue: as you write Windows code, assume

that each allocation might fail, or might succeed only at the expense of forcing the Windows kernel to go through a lengthy series of discards and moves.

In real mode, you can't leave memory blocks locked longer than necessary, either: you need to be nice to the other programs on the system. If you write a program that's a memory hog it'll slow down the user's whole Windows system, and your program will get a bad reputation. The price of following the rules for real mode is that your application may be a tad slower because it has to lock and unlock memory blocks all the time.

On the other hand, Windows uses the memory mapping hardware in standard and enhanced modes. If your program really needs a lot of memory, you might want to mark it protected-mode only with the **/t** switch to the resource compiler. If you limit your program to protected mode, the guidelines for memory usage are relaxed a bit: you can allocate and lock down all your memory when your program is initialized, and unlock and release it all when your window is destroyed. But don't allocate fixed memory blocks: these are different from movable blocks that have been locked, and might cause some fragmentation. Realize, too, that limiting your program to protected mode will limit your market to people with machines that can run standard or enhanced mode Windows on their computers. While that may not be much of a problem in the future, it will certainly keep you from selling your program to the huge installed base of 640K machines.

If you want, you can allow your program to run in all modes but optimize locking for protected mode. What you'd do is use the **GetWinFlags** function to determine the memory mode and set a global variable for your program. In protected mode you'd only lock a block once when it was allocated and release it once before freeing it. In real mode you'd unlock the block anytime it was not needed, and relock it whenever it was needed.

If you want to require Windows 3.1 to run your program, you automatically restrict your program to Standard or Enhanced mode: support for the three real modes was dropped in Windows 3.1, specifically to make life easier for developers. That doesn't change the marketing situation: somebody with an 8086 won't be able to run your programs. But it at least reduces the explanations you'll have to make—you can blame Microsoft—and it might even encourage people who want to run your programs to upgrade their machines rather than yell at you to support real mode.

Why Medium Model?

I advocate building most real-world Windows programs in the medium memory model. Small model is too restrictive, and compact and large models will not perform well in real mode. Medium model itself would be too restrictive because of the 64K limit on near data, except that you can augment the model with far data pointers and global heap allocations.

If you are building protected-mode-only programs, or programs that require Windows 3.1 or later, the arguments for medium model are somewhat weaker. In this case, large model programs may perform satisfactorily, although it is easier to get multiple instances of a medium model program than multiple instances of a large model program. I advocate medium-model multiple-instance programs rather than large-model Multiple Document Interface programs because they are faster, easier to maintain, and work better with DDE and OLE.

Unless your program is so small that all your code will fit into 64K, you'll need multiple code segments, which is the default in medium model. You can control the segment names explicitly with the **-NT** switch to MSC; in this way you can combine several object modules into a single segment. But most of the time, your problem wont be that segments are too small—it'll be that segments are too big. For dividing the functions in a module into multiple segments, you can use the **alloc_text** pragma in MSC. We'll discuss segment-size optimization and tuning at some length later on.

The default function call in medium model is done with long (far) code pointers. Far procedures are safe, but not efficient. When you're developing a program you should stick to the safe defaults; you can later optimize your code by changing those far functions which are called only within a segment to near procedures. If you switch to near procedures too soon, you may find that it's a burden to keep track of them when you tune your segmentation. Get things working correctly first; you can make them fast afterwards.

In order to access data on the global heap from medium model, you'll need to use far data pointers. This isn't especially hard—you just use the **LPSTR** type (defined in **windows.h** as **char _far ***) for strings on the global heap, or similar **_far** pointers for other data types.

Far data can get tricky when you need to use library routines. In the medium model almost all C library routines are built with far code and near data. The exceptions are special memory and string routines prefixed with **_f**, such as **_fstrcpy()**; these model-independent far functions were added in MSC 6.0 to support mixed-model programming. You don't usually need to use these functions from Windows, however; instead, you can use Windows kernel functions like **lstrcpy()**.

Many of the more specialized C library functions don't have far equivalents. You can do one of three things when you need to use one of these functions: roll your own far version from scratch, make a far version that does nothing but copy its arguments to local variables and call the library function, or copy the arguments to local variables before calling the library function. In practice, the last option turns out to be more convenient than it sounds, since where there is one library function there are often a string of them.