Open Computing ‘‘Hands-On’’ ‘‘PC-Unix Connection’’ Column: December 94

Windows Programming

In the final part of our three-part series, we show you that despite its commodity status Windows has some cool tools

By Tom Yager

When it comes to operating systems and graphical user interfaces, there is no group more demanding than programmers. As PCs replace larger iron, administrators and other in-house technical staffers are being called upon to create custom applications. Developers of commercial software, both vertical and broadmarket, are moving their wares to Windows in order to tap the huge market and give their users access to fast, affordable machines. And we ‘‘just users’’ of big-iron operating systems like Unix are finding ourselves without the formerly standard tools we once used to customize our machines to our liking.

Regardless of the size of the job, whether it’s creating a convenient pop-up menu or coding a huge commercial application, Windows can look like a dark and foreboding place. Petzold’s official Windows programming manual is a monstrous volume. You don’t look forward to having to learn all that just to get started. You’ve also heard that Windows is a lousy place to write code: unstable and with poor-quality tools.

It’s not that bad. As I’ve stated in the previous two columns, PC software--both DOS and Windows--has potholes. But we won’t pretend that Unix, or whatever else you’re using, is perfect. We all find workarounds for, and then forgive, our environment’s warts. That makes everything else’s warts loom like capitol domes.

Lest I be accused of liking Windows too much (as claimed in a few electronic nastygrams generated by my recent musings), I’ll start by accounting just what’s wrong with programming for Windows. Mostly, Windows is still a 16-bit operating system. That means memory management is not the cakewalk that it is in a nice, flat address space. Lacking true memory protection, the best Windows can do is try to figure out when your application breaks the rules, putting up a message (usually meaningless) as it drop-kicks your application from memory. Or, if you’re really good, you can make Windows itself freeze or die. You won’t have the same gigantic function libraries you took for granted in your old stomping grounds. And your tools? Forget it. PCs have their own, and you’ll have to learn them.

Now that I’ve sated the anti-Microsoft readers, let me address each of these concerns. First, regarding 16-bitness. Don’t expect me to wax lyrical about Chicago (now known as Windows 95), the mythical 32-bit commodity Windows. It keeps slipping, as Microsoft products are wont to do, and it’ll take time for applications to catch up even when it does appear.

There are a couple of workable approaches to current memory management hassles now. The first is to pick a good compiler. If you equip yourself with a decent modern compiler--Borland International,

Microsoft Corp., Watcom, and Symantec Corp. among them--it should have a code generator that automatically manages objects larger than 64K. Borland C++, with which I have the most recent experience (I use Microsoft Visual C++ as well), has a standard set of container objects that completely hides the vagaries of storage management. Just declare an array, pack it with stuff, and let it worry where the memory comes from.

Also making life easier on the memory management front is Win32S, a set of 32-bit extensions to Windows. This component is effectively a special subset of Windows NT, which is thoroughly 32-bit-adapted for Windows. The run-time support is free and freely redistributable. And again, good compilers will target Win32S. Take a little extra care to read up on the benefits and limitations of Win32S before you use it. The advantage is that Microsoft is promising that your Win32S code will be compatible with its current and future 32-bit operating systems.

Windows' infamous instabilities haven't been entirely banished to legend, but they are much more scarce than before. In some fairly extensive projects I've tackled recently, I've found the system's stability surprisingly good. While it lacks the virtual uncrashability of an operating system that employs true memory protection, advances have been made. Exception handling stands out among them.

Previously, when a mistake of yours--like writing outside an array's bounds or de-referencing an invalid pointer--caused Windows grief, you'd see it in a system-supplied requester window that let you know your application had already croaked. Now Windows can flag many once-fatal exception conditions for handling by your code. Special keywords--catch and throw--let you bracket code that might generate exceptions. You can respond to a nonfatal exception by continuing, in `setjmp(3)` style, at a safe recovery point in your code.

Unfortunately, exception handling isn't universally adopted. Your code may be calling out to libraries that don't handle exceptions, or don't handle them properly. For example, I had a recent project that called out to some third-party drivers that put moving video in a window. The driver code was buggy, and when it went in the bag, Windows flagged the error in my code. That made debugging loads of fun.

The Bat Utility Belt

You may have grown accustomed to stringing together collections of command-line tools to edit, manage, compile, and debug your applications. DOS, which provides the command-line side of Windows, is annoyingly single-tasking, so give up on your command-line approach. Also give up on C, if that's your pet language. You can code for Windows in C, but you'll be forsaking the Concorde for the bumpy country road: the Windows API. It's huge, musty, and all but impenetrable.

If you're a Uniphile, you can run your favorite tools under DOS. Mortice Kern Systems sells the MKS Toolkit, an assortment of Unix-alike commands for DOS, including the Korn shell. You can also get essentials like the source-code control system, Make, Yacc, and Lex from Mortice Kern Systems. There's nothing wrong with having these tools around--I swear by them myself--but plan to ease yourself into a more ''with it'' approach to Windows programming.

The state of the art, as espoused by dueling compiler giants Microsoft and Borland, is a combination of the C++ language with an integrated development environment (IDE). This environment brings your editor, debugger, graphical interface designer, compiler, and project manager under one cheery point-and-click shell. It's a radical departure from Unix cc, which may cause you to feel a little green at

the gills at first contact. But stick with it, because once you tune the IDE, it's really a productive place to hang out and make code. Both vendors also include command-line versions of their compilers in case you pine for the old days.

Debugging under Windows is an oddity. Both compilers' IDEs include integrated debuggers. Borland's graphical debugger is pretty good and includes expression evaluation, multiple watchpoints, and other needed features. As it is with Microsoft however, the real juice is to run your debugger in DOS. You can even give the debugger its own display, driven either with the serial port or by a second (monochrome) monitor driven by a cheap, secondary display card. At this level, you can get the works, including CPU-register watching and machine-instruction stepping.

Writing Windows programs using Borland, Microsoft, Watcom or Symantec compilers is simplified with standard class libraries. Microsoft has its own, the Microsoft Foundation Classes (MFC), which it bundles with its compilers and licenses to other compiler vendors. Borland went its own way, creating the Objectwindows Library (OWL). Both encapsulate the vagaries of the Windows API into sensible hierarchies of C++ objects. Instead of passing window handles around to display text and graphics, each window is an object with text and graphics-generating members. I found OWL thoroughly delightful after I quashed some initial anxieties. Curious, I scanned the library source code. After seeing what the Windows API-juggling OWL did on my behalf, I was glad I didn't have to write that code myself.

In addition to the graphical interface encapsulation, these compilers invariably include other useful objects: do time and date calculations with plus and minus signs; create dynamic arrays that grow and shrink themselves; make linked-list objects that allocate their own links; plus provide enough C-style function calls to keep you from having to hack your own porting functions. The Borland compiler's standard libraries have a surprising degree of overlap with Unix calls. My recent porting efforts went smoothly, with some time added by my elective updating of the code from C to C++.

Among other tools you'll find yourself using and appreciating is the Windows help compiler, which turns marked text into a nicely formatted, hypertext-linked, interactive reference. The viewer is the Windows help engine and is standard with every copy of Windows. So is Windows Write, a simple text edit-and-display tool that handles multiple fonts and text attributes. You can ship formatted README files with your application knowing that every user will be able to view them, even in foreign languages.

A wider array of networking options is available for Windows than for any other environment. Coming from Unix, you'll be pleased to know that TCP/IP is easy to get--even *free*--and Microsoft (surprised?) has endorsed the Windows Sockets interface. It's effectively a public-domain API you can use to write TCP/IP applications under Windows. It has already been used to port Mosaic, and it's also a good blend with Hummingbird Communication Ltd.'s X Window programming libraries for Windows. That's right, you can port X Windows applications directly to Windows where they'll run native on systems running the Hummingbird Exceed X Window System server.

One more interesting Windows programming tidbit is P-code. Microsoft has taken the age-old notion of meta-machine code--simple tokens that abstract groups of low-level CPU instructions--and given it new life. Microsoft's wildly popular Visual Basic programming language puts out P-code, and many commercial Microsoft applications execute mostly in P-code as well. Why? It's small. There are still a lot of four-megabyte Windows systems out there, and Windows programs are as hefty as many big-system applications. P-code cuts machine-instruction sequences to compact tokens. A tiny interpreter, which Windows invokes automatically, reads the tokens and does their bidding. The result is

a small, but not blazingly fast, application. Microsoft's C++ compiler will generate P-code for you if you like. As far as I'm aware, theirs is the only one; I'll let you know if I'm wrong about that.

How else will your application benefit? The Truetype scalable font system, as well as the nearly ubiquitous (but still not standard) Adobe Type Manager, give every Windows application access to scalable, rotatable fonts. The Windows Multimedia Extensions (standard since Windows 3.1), let you create applications with sound, video, animation, graphics, and even laserdisc control. The ease of the multimedia API, MCI (Media Control Interface), with its own built-in language interpreter that uses keywords like ''open,'' ''play,'' and ''seek''-has caused an explosion of applications that use sound and video elements. Contrary to its evil empire image, Microsoft commonly gives software away. Its Video for Windows full-motion video software is free, and it's standard practice for those selling Windows programming libraries not to charge redistribution royalties.

One more item: custom controls, or VBXs. Named because they were originally designed for Visual Basic, these libraries are really interface objects that know how to behave: a push button that displays an animated image, a spreadsheet object that computes formulas, a database-retrieval object that