Automating Windows with Perl

Scott McMahan

R&D Books
Lawrence, Kansas 66046

R&D Books
Miller Freeman, Inc.
1601 W. 23rd Street, Suite 200
Lawrence, KS 66046
USA

Cover art created by Robert Ward.

Distributed in the U.S. and Canada by:
Publishers Group West
P.O. Box 8843
Emeryville, CA 94662
ISBN: 0-87930-589-4

Foreword

It is an understatement to suggest that Scott likes computer books! Just visit his Web site at cyberreviews.skwc.com. Scott's passion for books and writing is impossible to miss.

As a matter of fact, it is because of his Web site that I first met Scott. On his site is a list that he calls *Essential Programmer's Bookshelf*, complete with in-depth, well-written reviews. These are the books that he believes no programmer should be without. A while back, he had some questions about one of my books that he was reviewing for inclusion in this list. One thing led to another, and since then I have always made sure that Scott received early review copies of my books as soon as they became available. Given Scott's writing talents, I knew that it was only a matter of time before he would be creating his own book.

Scott has written an enjoyable, readable book about automating Windows using Perl. At the same time that the computing environment is becoming easier for users, it is becoming more challenging for programmers. The tricks that we used yesterday look pass today and might be useless tomorrow. Of course, this keeps programming interesting. It's always changing. And Perl is part of that evolution. But, then, finding better ways to do things is what programming has always been about.

Herbert Schildt

Acknowledgments

Wisdom is the principal thing; therefore get wisdom: and with all thy getting get understanding.

Proverbs 4:7

This book is dedicated…

To God, for giving me the talent and calling to be a writer and allowing me to finally realize it.

And of course, to my mother, for all those years she sacrificed to get me an education so I could do stuff like this. She believed in me and encouraged me during all the times when things like writing a book seemed like an impossible dream.

Special thanks to Anthony Robbins (`www.tonyrobins.com`) and Casey Treat (`www.caseytreat.org`), whose books about understanding the meaning and potential of life have made me realize that I had a destiny to fulfill after all. The impact of these teachers on my life can't be underestimated.

Special thanks to Roger Batsel for his invaluable feedback on the book, especially in the early stages, and everyone else at SoftBase Systems for encouragement. Thanks go out to SoftBase President Gary Blair (and everyone else) for putting up with me writing the book in addition to everything else going on at the end of 1998.

Special thanks to George and Mike Steffanos of SK Web Construction for allowing me to host this book's web site at their server.

I want to thank R&D Books and editor Berney Williams (no, not the Yankee's center fielder) for taking a chance on an author who had never written a book but who thought he could! Also, thanks to *Windows Developer's Journal* editor Ron Burk for buying my first technical article. And, to Joe Casad, Michelle Dowdy, and everyone else at R&D who made my first book such an easy experience with their professionalism.

This book would not have been the same without the contribution of technical reviewers. With their comments and feedback, I have turned a collection of rambling essays on various Perl topics into a *book*. I extend special thanks to everyone who gave me feedback on the draft, including Randal Schwartz (yes, *that* Randal Schwartz!), Jim Lawless, Robin Berjon, Anthony Roeder, and Sergei Listvin.

Also, like *Woody's Office Watch* (`http://www.woodyswatch.com/wow/`) always does, a word of thanks to the people who made the music I listen to as I wrote this

book: Erin O'Donnell (http://www.erinodonnell.com/), Truth (http://www.truthmusic.org/), Chris Squire (http://www.nfte.org/Chris.Squire/), Harvey Bainbridge, Brand X, and a cast of thousands of bands from the early 1980s. Without the tunes, writing this book would not have been much fun! (I only wish Natalie Grant (http://www.benson.com/nat/) had released her debut album in time, so I could use my joke about her label, Benson Records, not being affiliated with Robert Guillaume. Alas!)

Table of Contents

Introduction

Pearl of delight that a prince doth please
To grace in gold enclosed so clear,
I vow that from over orient seas
Never proved I any in price her peer.

(from JRR Tolkien's translation of *Pearl*)

Who This Book Is For

- Do you have to administer a network with Windows machines on it? Is it your job to make a network run reliably? Are you frustrated by the lack of tools to automate system administration tasks in Windows? This book shows you ways to make your network run automatically.
- Are you a power user who wants your PC to do more tedious work automatically so you can concentrate on the important stuff? This book shows you how a few Perl scripts can save hours of your time.
- Is system and network administration your *second* job? Small Windows workgroups often don't have dedicated, full-time administrators, and you might have been given the administration job in addition to your regular duties. This book shows you ways to reduce the time you spend administering Windows.
- Do you want to make life easier for end users? Maybe you don't have time to handhold end users and want to create automatic processes to • make their computing easier by creating solutions that let them get their work done. This book shows you ways to create automated solutions.
- Have you had trouble finding information on Perl in Windows? This books contains many practical, hands-on projects showing Perl at its best in the Windows environment.
- Are you a hacker who wants a new frontier of almost unlimited potential? Perl can do many amazing things. This book may be the inspiration you need to get started.

This book is for all the real programmers and hackers out there. I wrote the book I wanted to read: a book that is immediately useful and practical, but that also has the background information, extra explanations, tidbits of history, and side trips to interesting places. It's not just another how-to book, but one I hope you can return to again and again.

What This Book Is Not

This book is not:

- A basic Perl tutorial. I assume that you know, or are willing to make a motivated effort to learn, the Perl basics. If you need help deciding where to go, the bibliography points you to many excellent sources. For this book, I assume you know Perl well enough to start writing useful programs. I do give significant tutorial information on new topics covered in this book, such as Automation, but I do not discuss Perl basics.
- A treatment of advanced, idiomatic, tricky, or clever Perl. My programs are the meat and potatoes of Perl programming. They are unspectacular, straightforward, and easy to follow. Plenty of resources exist for the clever stuff. Perl is a language that allows creativity of expression, but it is also a language that admits boring practicality. This book concentrates on the practical aspects.
- A Windows programming tutorial. I assume you either already know or are motivated to learn topics like Automation. Again, the bibliography points you to many excellent sources.
- A regular-expression tutorial. I only use regular expressions when I need to and do not pay much attention to them other than how they apply to the programs. If you are a Windows programmer who has never encountered regular expressions, I suggest a gentle introduction like *Learning Perl on Win32 Systems*.
- A Windows NT domain administration guide. If you are looking for module-by-module, function-by-function descriptions of everything you can do to administer a Windows NT domain using Perl (users, groups, drive sharing, security, etc.), this is not the book. I discuss a much higher level of system administration in this book.
- A place to find a discussion of PC serial-port programming in Perl. One amazingly frequent question on the Perl newsgroup is how to use the PC's COM port in Perl. I have no idea how to do it myself since I do not do that kind of prgramming, and Perl does not seem to be the best language for it.

Windows and Perl

Welcome to the first book in history that discusses using Perl to automate Windows. Emerging from my own hacking efforts with Perl under Windows, the topics in this book have been almost totally undocumented, or documented in a scattered and disconnected way. I share here what I've learned through years of using Perl to get my own system and network administration tasks done. I want to share with you some potential applications and some pitfalls of Perl I have discovered. Until now, Perl has

not been used to its full potential on Windows platforms, mainly because the techniques for using Perl in Windows have been obscure.

This book comes from my experience using Perl itself for over five years and Perl under Win32 for two years. The more I use Perl in Windows, the more I like it, and I encourage you to get started using Perl as an important part of your Windows problem-solving toolkit. Perl is even more powerful in Windows than it is in UNIX because not only can you do anything in Windows you can do in UNIX, Perl in Windows also lets you access Windows applications through Automation. I want to help you unlock the power and potential of Perl for automating and administrating Windows.

By training, I am a UNIX systems programmer and network administrator. I've used about every flavor of UNIX you can imagine.[1] I began my involvement with Perl (then in Version 4) back in 1993, when I took over the *Genesis Mailing List* on the Internet. I had found some barely functional mailing list software on the Internet, written in Perl, which I had to modify. I learned Perl by reading that code and the Perl manual page. *Programming Perl* had been released in 1991, but in 1993 I was a poor college student who did not have a lot of money to buy computer books, and I did not see a copy until later. The mailing list software was so inadequate that its shortcomings finally forced me to rewrite it from scratch, which I did as my senior research project in college. Since I had only one semester to complete this project, I used Perl as a rapid application development language (before the term RAD was well known).

Over the years I've become involved with Windows in addition to UNIX, especially with networks of heterogeneous machines. (The networks I administer have Windows clients, NT servers, UNIX boxes, and even mainframes.) I've come to appreciate and like Windows in its own way, particularly for its graphical capabilities and selection of off-the-shelf software.

The first time I ever used Perl for system administration in a non-UNIX environment came in the early days of DOS networking. On my first job, back in 1994, I administered an old DOS file server on a LANtastic network. I wanted to do a daily backup the first time the machine booted in the morning. I could not simply run the backup as part of the boot program, because the computer was extremely unreliable and had to be rebooted several times a day. (Its networking software, a collection of DOS TSRs and other components, was unstable.) I wanted the backup to happen the first time the machine booted for the day but not on subsequent reboots during the day.

I needed to write a program that could read from a file the date of the last reboot and check whether the date was the same as the date of the last backup. DOS certainly didn't provide any tools that could solve this problem. The batch file language would let you redirect the date into a file, but that was about all. DOS had no date comparison routines. So I immediately looked for a DOS port of Perl and wrote a program that returned a certain exit code if the date was the same day as the last reboot. I incorporated this program into `AUTOEXEC.BAT`, and my Perl-based backup routine ran until I finally threw out the old DOS-based networking software and upgraded to an intranet running TCP/IP.

How things have changed since then! The PC has gone from providing an unreliable network environment based mainly on proprietary networking protocols to a far more stable system with native support for TCP/IP. The operating system has left behind the world of DOS (which barely did enough to be considered an OS) and entered the world of 32-bit Windows, which is almost identical to UNIX in terms of operating system services provided.

Perl has come along for the ride. Once existing almost exclusively in the realm of UNIX, Perl is today the de facto standard "glue" language of both UNIX and Windows. Administrators use Perl to hold systems together and to perform administration tasks automatically.

No matter what your opinion of Microsoft, they have undeniably remained competitive in the last half of the 1990s by adding open industry standards to their products. A compelling argument can be made that all of Microsoft's success (or lack of decline) since 1994 can be traced back to their adoption of open standards in their products. Most of the new enhancements to their products since 1995 have been to add support for Internet-related features; since Microsoft's main revenue is from upgrades to existing products, the Internet support gave their customers a compelling reason to upgrade.

Almost lost, though, in all the hype and emotions caught up with Microsoft and the Internet, was Microsoft's first, tenuous step into the world of embracing open industry standards: they funded the port of Perl to Windows. This is probably the wisest investment they ever made. It turned NT into a serious operating system. Perl for Win32 first appeared in the Windows NT resource kit, although DOS versions had long been available.

Although Perl for Windows has been around for a number of years, I have seen that the potential it offers is largely that: still potential. People have yet to begin taking full advantage of what Perl has to offer, which is where this book comes in. In these chapters, I discuss real Perl projects I've written to help make my life as a network administrator easier, and I describe how to integrate Perl more deeply with Windows programming and administration.

What Is Perl?

Although this book is not a tutorial, I would like to begin by looking at why Perl is such a compelling language for software development and system administration. What is Perl and why do so many people use it?

The most concise and expressive definition of Perl I have been able to come up with is:

> Perl is the portable distillation of the UNIX philosophy.

Perl is the smallest, most pure embodiment of the UNIX culture and philosophy. The design that motivated UNIX was a philosophy of creating tools that perform specific jobs well. UNIX carried out this design with commands (small, standalone programs), but Perl mimics the same idea with its built-in functions and modules. In a UNIX system, the emphasis is on using plain text files and processing them using regular expressions. This text orientation was a reaction to the myriad inflexible file formats found on mainframes and larger systems of the time when UNIX gestated, but it is equally important in today's world of uneditable, proprietary binary file formats. In the Perl language, you can find a complete microcosm of all that is powerful about UNIX. Ironically, Perl—like the Emacs editor—violates the UNIX philosophy of small,

minimal tools because it is so complete. Perl has, though, created a portable package that distills and embodies the UNIX philosophy compactly enough to port to other platforms. Because Perl is one cohesive whole, Perl eliminates many of the small, annoying inconsistencies found among the UNIX utilities and their various clones.

Many people who are exposed to Perl for the first time without ever having had any experience with UNIX find Perl baffling. Perl is easy to pick up if you have a strong background in UNIX utilities like `grep`, `awk`, and `vi`. If you don't know these utilities, however, learning Perl will be difficult because Perl is a product of a unique culture. I encourage you to spend the time to learn Perl, even if you are not familiar with its UNIX roots. Perl is one of those rare languages and tools (like C and Emacs) that repays you exponentially for the effort and time you put into learning it. If you don't already have a UNIX background, it may take you a while to learn Perl, especially when it comes to regular expressions. Learning the concepts Perl borrows from UNIX will be beneficial to any computer professional, since UNIX's concepts have permeated all aspects of computing. Your Perl experience will help not only with Perl, but most likely with everything else you do with computers.

I learned Perl largely through reading source code. Fortunately, since the time I started using Perl, the availability of Perl educational material has increased substantially in quality and quantity.

Unlike most software for Windows, Perl is free of charge. You can download it today and start using it. Perl is covered under either the GNU software license (also known as the "copyleft") or the Perl-specific artistic license. Because some people have trouble with the copyleft license, the more relaxed artistic license is included. See the Perl distribution for details. Perl is one of the oldest free software packages on the Internet.

Perl comes with a *lot* of documentation. The latest ActiveState port (discussed later) automatically installs all the documentation in HTML format, so you can view it with your regular web browser. Perl's documentation is meant to be used as reference, and I encourage you to explore it and become familiar with it. Books such as mine assume you will turn to the reference documentation for clarification of anything that gives you trouble. The completeness and usefulness of the documentation can't be overestimated. If you need tutorial information instead of reference, consult some of the books listed in my Bibliography.

Perl is both an interpreted and a compiled language. Scripts are interpreted at runtime, but not like a traditional line-by-line interpreter (such as old BASIC interpreters). Perl code is parsed into an internal compiled form before being executed, so it does not have the severe runtime performance penalties normally associated with traditional line-by-line interpreters. Perl code is usually quite fast, and Perl's interpreted nature is usually not a concern since Perl programs don't often do tasks for which code execution speed is the major bottleneck. Perl code typically does file I/O, regular-expression parsing, operating system interfacing, and other tasks that would not be any faster in a compiled language.[2] Perl's interpretation is done at runtime, and the program's source is taken from platform to platform, which gives it an advantage over Java's byte-code interpretation because if something does not work in a Perl program, you can hack the Perl source on the spot. You can't change the Java byte-code because you don't have the source available.

Perl:

- Automates the tedious
- Executes noninteractively
- Has all the tools you need
- Provides easy access to the operating system and its configuration

Perl allows you to automate tedious tasks. If you perform a step-by-step procedure often, chances are you can write a Perl program to automate it.

Perl has all the tools you need; it is a complete package. The UNIX command line doesn't port to other platforms well because UNIX is a collection of hundreds of utilities strung together by shell scripts. Perl is a single executable with the functionality of the separate utilities built into it. Perl stands nearly alone in this way. (GNU Emacs is the only other comparable example I know of.) Few scripting languages can exist outside of their native environments, because they rely so heavily on the commands provided by the underlying operating system. Perl insulates you from the operating system and allows you to write self-contained programs.

Perl allows you to easily access important configuration settings on the system. In UNIX, these important settings are generally found in text files. (The famous example is `/etc/passwd`.) In Windows, these settings are held in text files, contained in the Registry, and accessed through Automation.

There is more to Perl than I could ever list here, but let me discuss a few other significant reasons why I use Perl. Most importantly, *Perl does not limit your thinking*. All computer programs are, at their core, solutions to problems. Most of the other programming languages I've seen were designed by programmers who were thinking of the machine language the compiler would have to generate (C is certainly the prototype of this variety) or designed for some specific goal the inventor had in mind (for example, C++ and Java were both meant to look like C, and Pascal was designed for educational purposes). In most other computer programming languages (except those specialized for a certain problem domain, such as Prolog), you must invent a solution to a problem in terms of how the language itself works, not what is most natural to your brain. Perl allows you to state the solution in any terms that come to mind. Perl was invented by a *linguist,* Larry Wall, whose primary goal was to create something usable by programmers. Anything you can do in Perl can be done in more than one way. In most situations, you can code the solution in the same way you mentally conceptualize the problem. You do not have to adjust your brain to think in the same manner as the person or people who wrote the language. I might add that one factor of C's success is that the people who wrote it captured the mental processes of the people who would be using it! This is in sharp contrast to other languages in use at the time, which had various design goals (COBOL for business, BASIC and Pascal for education) but were not really designed for the day-to-day use of programmers. Perl does C one better: instead of just narrowly focusing on systems programming, Perl supports solutions to problems in any problem domain.

Writing programs in Perl offers many advantages over writing in any other language, particularly within Windows:

- Complete integration of file processing with the rest of the language. Too many other scripting languages treat files as something either separate or tacked on to the language. (Cf. awk and batch files.) Perl allows you to naturally express file processing algorithms in ways that make programming easier.
- Natural integration of math into the language. Shell programming languages have unholy pacts with external programs (with names like [) to allow you to do math.[3] DOS batch files have no math capabilities at all. Math is a natural and inescapable part of programming, and Perl makes math extremely easy.
- Built-in report formatting. No shell or batch language I've ever seen has built-in report formatting. You can generate sophisticated reports very •easily. Perl reports are most naturally plain text, but it is also possible to generate HTML reports using Perl reports.

- Built-in systems programming capabilities. The Perl language, unlike most other scripting languages, rivals the power of C. Most scripting languages allow you to run external commands, but Perl also provides all the capabilities C provides for calling the operating system. Try getting a file creation date in a batch file, or try using the DOS batch language to access Automation components. In Perl, getting information from the operating system is usually easier than it is in C. The flip side of Perl's integration with the OS is that you do not need to rely on external programs. Batch files, REXX programs, and shell scripts all require certain external programs in order to do anything useful. Perl does not, and Perl programs are more portable as a result.

Perl has a logical, familiar syntax. As important as scripting languages are, most of them are very unfriendly and have weird, hard to understand syntax. (Take the C shell or the Windows batch file language; a catalogue of their weirdness is beyond the scope of this book, but, trust me, they're *weird*.) Perl's syntax is designed to be usable to anyone who knows C, and that's most administrators and computer programmers.

Perl is the original Rapid Application Development (RAD) language. Perl was a RAD tool before RAD became such a widespread concept. Perl is one of the first truly effective RAD tools. Because Perl handles things for you that a language like C does not, writing programs in Perl is generally much, much faster. Perl automatically handles all memory allocation and deletion for you. You don't have to write pages of code to allocate and free memory. Perl dynamically allocates and resizes strings, arrays, and other data structures for which C programmers must reinvent the wheel every time a C program is written.

Perl's Influences

Perl is the product of certain influences, and it is worthwhile to examine these and see how they add to the whole of Perl. Perl is greater than the sum of its influences because it blends them so seamlessly.

Perl's syntax resembles C more than anything else, except for the regular-expression syntax and operators. Although Perl borrows heavily from C, it omits most of C's extreme low-level features. Perl intentionally leaves out the low-level, almost assembly-like manual memory allocation and pointer manipulation of C. (These drawbacks in C led to the creation of C++ and finally to the Standard Template Library.) Most C programming time is spent, after all, reinventing the various wheels over and over by writing similar memory management code. Perl manages memory for you. Perl includes the familiar syntax of C. The C language has been the foundation for every[4] major language with widespread adoption since the late 70s/early 80s. (Compare Perl, Java, JavaScript, and C++ for starters. All use C syntax as their basis.) Once C became entrenched, new languages seem to have to look like C to be

accepted by programmers, who don't want to learn the radical syntactical differences of non-C languages.[5] Perl is immediately familiar to a professional programmer, since C is the *sine qua non* of modern programming. Perl's acceptance also seems to be strongest in the generation raised on C, those who became programmers in the 1980s and 1990s.

Perl borrowed the idea of being a complete language from C as well. Before Perl, "scripting" languages often had woeful support for calling the operating system directly (as C can do easily) and for mathematics. For example, the support for doing math is complex and ugly in the original UNIX shell programming language. Perl was really one of the first complete high level languages that was completely flexible with regard to mixing text and numbers.

Perl borrows from awk the concept of matching input lines against regular expressions. Regular expressions themselves are an integral part of UNIX (although almost totally unknown outside of it in my experience), and regular expressions form the core of commands such as `awk`, `sed`, `ed/ex/vi`, `grep`, and others. (Indeed the name `grep` comes from the old `vi` syntax, `g/re/p`, where `re` is a regular expression.) awk took the idea further than the other tools and made regular expressions the basis of a scripting language where a block of code was attached to each regular expression and executed if an input line matched the regular expression. Perl supports this style of coding but relaxes awk's restrictive implementation. For awk to be useful, it had to be combined with shell scripting tools (in all cases other than the most basic text processing).

Also from awk, Perl gets the idea of *hashes*. The hash is a data structure that allows a string index into an array. In awk, this is the *only kind* of array (even numeric indices are treated as strings). awk itself and earlier versions of the Perl documentation call hashes *associative arrays.*[6] (You can date when a programmer learned Perl by which terminology is used. I learned Perl back when the admittedly cumbersome term *associative arrays* was current, and I must force myself to use the term *hash* in its place.) Perl uses the term *hash* because that is the underlying data structure used to implement associative arrays. (This data structure has also been called a "dictionary" or a "map" in some circles. If you are familiar with the Standard C++ library, it has a map.) The hash is one of the most useful data structures in computing, because it allows you to express the natural idea of a tag or key that points to a much larger chunk of data. The need for this organization of data comes up time and time again in the solution to programming problems. Most, if not all, modern languages (particularly object-oriented languages) have some form of hash. Languages without hashes are painful to program in, since generally at some point you find yourself reinventing a hash from scratch, and if the language does not support the data structure, that invention is usually cumbersome.

From LISP, Perl brings in the idea of lists. The LISP connection with Perl is often overlooked because Perl code doesn't look much like LISP code. The LISP language is a dying language, and LISP hackers are a dying breed. LISP has fallen out of mainstream usage, probably in large part because of its counterintuitive inside-out syntax. (All LISP expressions are of the form: `(operator operand1 operand2)`. Most people want to work with a language that uses normal infix notation to express ideas.)

Perl takes LISP's central idea, processing lists, and gives it a C-like syntax that is much easier to use. The concepts of list processing are so useful that they are a core part of Perl. You can iterate through a list (with both `foreach` and `array` notation in Perl), split a list into parts (with `split`), join one list with another (with `join`), and take sections out of a list to form a new list (with `slice`). Perl makes lists an integral part of the language. In Perl, a list is an array. A list is also a stack. A stack in Perl is just a list you choose to use as a stack. The equivalence of these similar data structures means you do not have to be constantly converting one to another. Most languages treat every data structure as separate. In Perl, the same data can often be treated as a list, array, or stack depending on what you are trying to do with the data. It is very natural to use stack operations to create a list, then list operations to rearrange the list, and then array operations to iterate over the list. Perl also makes it easy to get non-list data into a list. Few languages, if any, have such support built into them. Perl allows a program to read each line of a file into a list, using only one line of idiomatic Perl code.

Perl has improved upon the string processing facilities in C, C++, and Java. One of the biggest drawbacks to using a language like C is its primitive string-handling capabilities. Strings in C must be managed byte-by-byte instead of as an atomic[7] data type, with memory management by the programmer. Java, which could have done what Perl did and make text processing easy, instead inflicted `String` and `StringBuffer` on programmers. Java's woeful string-handling capabilities have always surprised me, since it was a modern language designed to build on both the successes and failures of C and C++. I would have expected them to integrate powerful string processing into the language more closely. Even the new Standard C++'s string data type doesn't come close to Perl's support for strings (although it is still better than C's or Java's). Perl is one of the few languages available that makes string handling easy and natural. Perl derives this string-handling capability from the spirit of awk, but with many improvements. (awk, for example, does not have the string/list integration of Perl, although awk does have a primitive `split()` function.)

The Perl language has strong similarities to the English language, because it has borrowed from many other languages and melded what it borrowed into something that is greater than the sum of its parts.

Perl Doesn't Exist in a Vacuum

Perl doesn't exist alone. It is part of a greater whole and must by its nature (as a glue language) exist in a system that includes other technologies. In UNIX, Perl is one of many[8] UNIX commands. It is difficult, if not impossible, to use Perl without having to interface with the rest of the system.

In Windows, too, Perl doesn't exist alone. In this book, you encounter VBScript (Visual Basic, Scripting Edition), VBA (Visual Basic for Applications), C++, Automation, and many other Windows languages and technologies. Using Perl by itself is possible, but to truly unlock the power of Perl on the Windows platform you need to let Perl interact with the rest of Windows.

I assume you know enough about Windows to have a conceptual understanding of VBScript. The advanced chapters assume you've done some C++ programming. If not, you can consult the books listed in the Bibliography for more information.

If you are unfamiliar with Visual Basic, learning enough to get by would be helpful, if only because most Automation examples have been written in Visual Basic. VB is the native language of Automation client examples. Some familiarity with Visual Basic is required for reading any of the literature about Automation. If you know any object-oriented programming at all, and know BASIC, VB should be trivial to learn. If you already have learned Perl, VB is easy to learn.

Common Ground

Before I get too far into the book, I will describe some common terms.

When I say *administrator*, I'm talking about either a system or a network administrator. The line between a system administrator and a network administrator has blurred, especially in small shops where Windows would most likely be used.

By the terms *Win32* and *Windows* I mean a 32-bit Microsoft Windows operating system such as Windows NT, Windows 95, or Windows 98. For the Perl programmer, there is almost no difference between the 95/98 variety of Windows and NT. The differences are at a much, much lower level.

Indeed, for this book's purposes, there is no difference between Windows NT and 9x. All of the code in this book will work on either. (All of it was developed under Windows 98.) The only real difference between NT and 9x lies in some of the Perl modules you can use for network administration on NT that you can't on 9x. Windows 9x is a client, and you can't do a lot of the administration tasks on Windows 9x you can do on an NT server machine. Much information on NT administration is available in

existing books, and I do not try to duplicate it.

> As this book was being written, Microsoft announced that the Windows 9x operating system had come to an end with Windows 98 (and reversed this announcement before the book was published) and that Windows NT v5.0 would be renamed Windows 2000. By Windows NT, I mean Windows NT 4.0, with at least Service Pack 3. I have not used Windows 2000 yet, but I expect this book to apply to Windows 2000 as well.

In this book, I do not deal with 16-bit programs of either the DOS or Windows 3.1 persuasion. 16-bit technology is dead. It's time to bury it and move on.

The DOS prompt is an extremely misleading term, and to avoid confusion, I use the terms *command prompt*, *console window*, and *console prompt* to refer to what most people call *the DOS prompt.* Even though you can run old DOS applications from this prompt, you can also run Win32 console mode applications like the Perl interpreter, and I feel calling it a *DOS* prompt is grossly inaccurate.

What I call *Automation*, with a capital A, has also been known as OLE Automation, and by the time this book is finished it could very well be known by a different name. Microsoft presents a moving target with its constant technology-renaming initiatives. If you look at the history of what's now called *ActiveX*, the underlying technology is almost unchanged since OLE 2.0 came out, only the terminology has changed.

Perl programs have been called by many different names. I'll call them *programs* for the most part. A synonym is *scripts*.

A point of confusion, particularly among UNIX types coming to Windows, is the path separator character. Inside the DLLs that implement Win32 (and this observation even applies back to DOS), the / (slash or forward slash) character and the \ (backslash) character are interchangeable. The OS's file I/O and directory I/O routines do not care. If you don't believe me, open Internet Explorer and type a file:// URL and use / as the path separator. (E.g., `file://c:/windows/.`) It works. Only the shell (`COMMAND.COM` in Windows 9x and `CMD.EXE` in Windows NT) uses \ exclusively as the path separator. (Other shells, like the Cygnus BASH, do not.)

The reason why the command processor has to use \ as the path separator goes all the way back to DOS 2.0. Microsoft, having been heavily involved with the XENIX operating system, wanted to add a UNIX-like hierarchical file system to DOS. Before DOS 2.0, a directory tree was unnecessary, since the PC itself had only a floppy disk as disk storage. You couldn't put enough files on the disk to need a hierarchical file system. But with DOS 2.0, new fixed disks (later called hard disks)[9] were appearing, and they had the ability to grow to gigantic sizes like 10MB. (Compared to a capacity of 360K, 10MB was large.)

Unfortunately, DOS already used the UNIX path separator (/) as the character that introduced command line options (e.g., `DIR /W`). This usage existed because the original DOS 1.0 was closely modeled on CP/M, the (then) dominant 8-bit operating system. The new 16-bit operating system, DOS, tried to be similar to CP/M so it would be friendly to both CP/M users (then by far the majority of microcomputer users) and programmers (by being as close to CP/M as possible so programs could be ported easily). This similarity included using CP/M's command line syntax. Faced with a dilemma, Microsoft chose to go with backwards compatibility and keep the option character the same while changing the path separator to \. This decision will go down in history as a truly boneheaded choice, but many decisions appear boneheaded in hindsight. The important thing is that this change is true only in the *shell*. In the operating system itself, it doesn't matter which path separator you use. DOS and Windows have always been able to use either internally.

This book uses / for the path separator unless the pathname in question is being passed to the shell. No good reason exists for using backslashes internally in any program, Perl or otherwise. Perl (like C) treats the backslash as a special escape character, making its use as a regular character tricky.

In this book, I do not use academic footnotes to cite every book I mention. I list my sources in the bibliography at the end of this book.

Got Perl?

You can finally get a single, unified Perl for Windows. Starting with Perl v5.005, both the UNIX and Windows versions of Perl come from the same code base. Perl has been fragmented in the past, with the Win32 version generally lagging behind the "official" UNIX version but supporting extensions the UNIX version did not support. Perl v5.005 will be more uniform.

I included a Perl interpreter on the CD-ROM, but you should check for a newer version, because updates come out so often. Download the latest and greatest interpreter available.

I highly recommend you download Perl from ActiveState. With the introduction of 5.005, ActiveState provides a precompiled binary distribution of the official Perl 5.005 source, packaged with a native Windows setup program. This setup will configure your system properly to use Perl. The URL is `http://www.activestate.com`.

If you need more than just Perl, there's the *Perl Resource Kit* (PRK), published by O'Reilly. The PRK contains an official, supported version of Perl and many extras you won't find in the free distribution. The highlight is the graphical debugger, which in my opinion is worth the PRK's price, *if* you can afford it. Mortice-Kern Systems sells an MKS Toolkit, which I have never used and do not possess, that also comes with Perl. This MKS Toolkit is a commercial product that provides all the UNIX utilities for Win32 systems. All of these commercial packages are strictly optional extras. Everything in this book, and everything you can do with Perl, can be done with the normal Perl distribution.

This book will not discuss how to compile Perl yourself. I feel this information is a moot point for the vast majority of Perl users. Several Win32 binary distributions exist, and unlike UNIX, a single Perl binary will run on (almost) all Windows systems. You do not need to recompile and tune the binaries like you would on UNIX. The only reason to recompile Perl yourself is to change the Perl interpreter in some way,[10] and if you can do that you're programming at a level that is much more advanced than this book.

Install and Run Perl

Installing Perl 5.005 is painless and easy. I highly recommend downloading the ActiveState distribution and running the provided setup program. Manually installing Perl is not recommended. The ActiveState setup program will also properly configure Perl for CGI programs. The native Windows setup program is so automated and simple that I am perplexed by all the questions I see from people trying to get Perl running. I'm not sure how it can go wrong.

Once Perl is installed, it's time to run a program. How you approach this task depends on whether you are a Windows user coming to Perl for the first time or a UNIX user coming to Windows for the first time and bringing Perl with you.

If you are a Windows user, I strongly suggest that you run Perl programs from a console prompt window. The natural tendency for Windows users is to want to run Perl programs by associating the `.pl` extension with Perl in Explorer, and double-clicking on Perl programs to run them. This is not a good idea, however, because Perl has been designed inside and out with the assumption that it is going to run in a command line environment. It doesn't adapt well to a GUI environment like Explorer.

The two most common problems you encounter when running Perl from Explorer are:

1. Perl programs tend to want command line arguments, and it is difficult to give these through the Explorer interface. Particularly, the kind of command line arguments that change from run to run are impossible to use in Explorer without getting a third party add-on. The kind that are set once and are permanent thereafter are not as difficult.
2. The command prompt window tends to close after the program runs, which makes inspecting output difficult. To make the window stay visible after the program ends, you must create a PIF file for the Perl interpreter on *every machine* on which you'll run the program.

My advice is to always use a console window to run Perl programs. Go to the directory where the Perl program is, and run the interpreter from there.

If you are a UNIX user, the command prompt is the natural way to run Perl programs. If you are using a UNIX shell clone of some sort (like bash or tcsh), you can skip this discussion. But if you are going to use the Windows shell (`command.com` under Windows 9x and `cmd.exe` under NT), you must know that there is no equivalent to the *magic number* first line of UNIX shells. That is, files starting with the characters `#!` (0x2321) are considered scripts in UNIX, and the path that follows the characters `#!` on the first line is taken as the path to the script interpreter (for example, `#!/usr/local/bin/perl`). The Windows shells have no concept of this syntax.

One method for invoking Perl in Windows is to do what I do: explicitly type the command `perl`, on the command line:

```
C:> perl somescript.pl
```

It is also possible to create a batch file that is both a valid batch file and a valid Perl program. This technique also exists in UNIX (where a valid shell script is also a valid Perl program, for those UNIX variants that do not support the `#!` syntax), but it is much less common in UNIX. The batch file is invoked as a batch file by the command interpreter, and then the same file is passed off to the Perl interpreter by the batch file.

```
@rem='
@echo off
perl %0.bat
goto BatchEnd
';

# Perl code here
print "Hello world!\n";
__END__
 End of Perl code

:BatchEnd
```

In this idiom, the Perl code is ignored by the batch file, and the batch file code is ignored by Perl. The `@rem='` line is the key: to a batch file, this is a remark, which is ignored. (The `@` character is a special signal to suppress output of the line on the console, and `REM` introduces a comment.) The batch file runs line-by-line until the Perl interpreter is invoked on `%0`, the name of the command itself, in this case the batch file. Then, after the Perl interpreter exits, the `goto` in the batch file skips over the code until it reaches the label `BatchEnd`. (Unlike Perl, the command interpreter does not preparse the entire batch file; it instead parses line-by-line, so the Perl code is simply ignored.) To Perl, the `@rem=` is an unused variable initialization, initializing the array `rem` to a list of one item, a multi-line string containing all of the batch file starter code. The `__END__` line is considered the end of the Perl program, and everything that comes after it is considered a comment.

I've switched back and forth between Windows and UNIX for so long that I use the explicit Perl interpreter invocation on both types of systems.

About the CD and Code

The CD that comes with this book has all the Perl programs and other programs developed in this book.

The Perl programs, for the most part, are so short that I have not attempted to make them run under `strict` and `-w`. The cron clone and the backup program are both of such length and complexity that I have made them run cleanly under the full `strict` and `-w` scrutiny. The other programs are short demonstrations of specific ideas, and I have not attempted to turn them into production-quality programs. I see the rest of the programs as building blocks that are starting points towards bigger and more complete programs, which would then be made to run under `strict` and `-w`.

> Perl gurus recommend that all Perl programs be run with the command-line switch `-w` (which turns on compiler warnings). The gurus also recommend you place the line `use strict` at the beginning of the program. `strict` and `-w` together are powerful debugging tools that take away many of the dangers and pitfalls of a free-format interpreted language. Sometimes this is a good idea, especially for larger and more complex programs, but for very small programs, the overhead gets in the way. I recommend any code you use in a production environment, i.e., for something important like running your network, be subjected to the scrutiny of `strict` and `-w`, and that you heed the warnings you receive from `strict` and `-w`.

The code has been tested with the following development tools:

- All the Perl programs were written and tested under Perl v5.005 (using ActiveState Build 504). They should work on all higher versions, but they will not work under lower ones.
- The C++ code was all compiled under Visual C++ v6.0. The code should work under Visual C++ v5.0 as well, but it has not been tested under v5.0. The C++ code probably will not work under pre-v5.0 Visual C++.
- The Delphi code (what little bit of it there is in the book) has been tested under Delphi 3.
- The Visual Basic for Applications code was all developed and tested under the various Office 97 applications. It will not work under any previous versions of Microsoft Office, but it will probably work under later versions.

I've included a bonus program on the CD from my Essential 98 suite, which will be useful with some of the Perl programs developed in this book. Everyone who buys this book gets a free, registered copy of Run Control 98. This program is the missing piece of Windows 9x (it works under both 95 and 98) that allows you to control which programs are run automatically when the machine boots.

The Book's Web Site

```
http://autoperl.skwc.com
```

The computer industry moves too fast for books alone. The official *Automating Windows with Perl* web site will keep you up to date with late additions, corrections and errata, downloadable source code, and more.

The site is sponsored by SK Web Construction, creators of professional and exciting web presence for businesses.

Author's Note on Modules

This book has a patchwork approach to Perl modules that deserves explanation. Most of the people who read early versions of this book said, "Why didn't you use such-and-such module here?"

Perl has a large library of prewritten modules that rivals the size of any other language module library I have ever seen, even PL/I's.[11] These modules do just about anything you can think of to do. In Windows, a module allows you to use Automation servers. Every Internet protocol you would want to use is wrapped up in a module. Most common tasks are encapsulated in modules, like recursive processing of the directory tree. In short, Perl modules have, in a very short time, become one of the most wildly successful examples of code reuse in the history of computing.

At this point in my Perl career, I use Perl 5.005 exclusively and actually use modules a lot. I enjoy the aspect of code reuse. Anything that saves me from writing extra code is welcomed. It has taken me a certain amount of time to get comfortable with modules, and this book tends to show me in a transition phase because the code I present is usually several years old. I have two reasons for being slow to catch on. First, when I learned Perl, I learned the older Perl v4 *sans* modules. I learned Perl v4, and I tend to write Perl v4-ish code even today.[12] The use of modules is not natural to me, and it's taken me some time to get used to them. Second, the Win32 port of Perl has been notorious for not supporting standard Perl modules until the recent v5.005 release. Even if I had wanted to use modules, I could not in most cases.

The only difficulty is that most of the base of Perl code I have written predates the availability of modules. I have attempted to rewrite several of my programs to use modules. My `tare` command appears in both its pre-module form and in a form that

uses the standard `File::Find` module. My `graburl` function has been totally rewritten to use the LWP module. It formerly used raw sockets.

The most significant chapter not to be rewritten to use modules is the chapter on CGI. Recently, `CGI.pm` has become the standard Perl CGI interface, but my code long predates that. I wrote my CGI code from scratch during a time when many competing Perl CGI solutions were available. In one respect, I like the power and standardization of `CGI.pm` (even though I have never had time to rewrite any of my code). In another way, though, I like my code. It has served me well in less than optimal situations. I do a lot of CGI programming for servers where I have little or no control over what is installed on the final machine. I often do offline development of code on my Personal Web Server and then upload that code to UNIX servers. Other people have ported my programs to other servers without my being involved. The streamlined and self-contained nature of my code has helped me with this aspect of my CGI programming.

A Brief Note on Anti-Microsoft Sentiment

Writing a book combining open-source Perl and proprietary Windows is asking for trouble. Culturally, the two are worlds apart. But if you are looking for any fuel to feed your anti-Microsoft fires in this book, you won't find it. Outside of this introduction, I will not discuss Microsoft's position in the computer industry or their competition. All these issues are, to me, irrelevant and blown out of all reasonable proportion. I am only discussing them here to be (hopefully) a voice of reason and to present a viewpoint that I have not seen discussed.

Anyone who wants to understand the entire issue of Microsoft, monopolies, competition, and so forth in the computer industry should read Paul Zane Pilzer's *Unlimited Wealth*. Pilzer's argument is that we have transformed our economy from a zero-sum, scarcity model (with industrial products manufactured out of raw materials, which would always be scarce and limited) to one of unlimited potential with no limits to the amount of wealth that can be generated. He argues that it is no longer the case that someone must lose for someone else to win, because we are in an information age where there is no scarcity of resources. *Anyone* with a good idea has a reasonable chance of being successful. Interesting to me, Pilzer wrote his book *before* the Internet explosion of 1994, and everything he writes has been proven true by the Internet's widespread adoption and the kinds of success stories it has generated. Because his is not a "computer" book it has unfortunately been overlooked by people trying to understand software.

If Microsoft has a fatal flaw (and I'm not sure it does since it seems to be able to reinvent the company in ways no other company has ever done), it is the fact that its way of looking at all things related to software is the old zero-sum mentality. Every scrap of evidence concerning Microsoft I've either read or witnessed personally testifies to the fact that its business philosophy is built around the view that they can

only win if someone else loses. It brings this mindset to software marketing and sales. The single exception has been Microsoft's getting serious about the Internet, which is a sign the organization can change if it understands the need for an *ideological* change and not a superficial one. I have read many biographies of Bill Gates, and it seems to me from all I've read that he looks at the world through zero-sum, scarcity eyes. I do not think it insignificant that he is having less to do with the business affairs of Microsoft at a time when massive ideological changes will be needed to keep Microsoft viable. But only time will tell.

The competitors who are behind the lawsuits I have read about (particularly those alleging that Microsoft has a monopoly) are driven by exactly the same zero-sum mentality. I am completely convinced that the people who are suing Microsoft and trying to bust Microsoft into small, impotent companies would not be able to compete either if they were given a completely level playing field. They're too locked into the zero-sum mentality to ever be successful in the new world. They would just try to replace Microsoft. By analogy, the zero-sum mentality is Sauron's ring,[13] and anyone who uses it to depose Sauron will simply become a new Sauron. Nothing will have changed. These competitors demonstrate they are locked into the zero-sum way of doing business by the fact that they believe Microsoft has a monopoly and is capable of controlling them. The only way they can win is if Microsoft loses, and that will have to be done by making Microsoft's success illegal, which is precisely a zero-sum way of approaching the problem.

Who's winning? Interestingly, neither Microsoft nor those trying to sue Microsoft are doing anything exciting. For all that they have been in the news, the companies haven't done anything interesting in a long time. Where is the excitement and innovation in the computer industry? The true winners are the people who are ignoring the old zero-sum game and pioneering new areas of growth that have nothing particularly to do with the old territory. The only area of any significant growth or even interest in the technology landscape of the late 1990s is open source software, which is the living embodiment of everything Pilzer talks about in his book.

All the study I've done of this issue points to abandoning the zero-sum mentality. Microsoft has a "monopoly" that will exist only as long as people believe it does and continue to plan their lives and companies around its existence. Where in the Perl, Linux, GNU, and other movements was there any acknowledgement of Microsoft at all? They didn't set out to compete with Microsoft, and their success is not measured by how much market share they take from Microsoft. The computer industry is so wide-open that there's room for everyone, and there is absolutely no way to stop innovation. The only thing that stops people is old ways of thinking that no longer apply.

I don't believe that the coincidence that open source is free of charge is what makes it successful, at least not entirely.[14] Any software that follows the unlimited wealth idea will be successful (as long as the software is worth something). Open source just

happened, consciously or unconsciously, to embrace the unlimited wealth model, and it has been one of the first product categories to do so. I am personally not against commercial software. I have several programs I paid good money for; they are worth what I paid, and my daily computing would not be much fun without them. The general trend I see with open source software, though, is that open source is the best software regardless of price. Money can't buy better software. If it could, you could sell it, but it can't. The open source software is simply the superior software in any given category. For all the ideological battles surrounding open source, in the end I believe simple excellence will be the deciding factor in the widespread adoption of open source software by the entire computer industry. People are waking up to the fact that reliability and quality are the most important aspects of software.

Notes

1. I've used Ultrix, XENIX, Minix, Sun OS, Solaris, HP-UX, AIX, 386 BSD, and Linux--and probably others I've forgotten.
2. Often, people ask if there is a "Perl to C" converter, because they want to compile Perl for more speed. Depending on what the program does, though, compilation in the traditional C and C++ sense may not help the runtime speed. For example, a program that spends all its time reading from a socket and updating a database via an ODBC connection will likely see little or no speed increase, since the network and database connections will be the major bottlenecks in the program (unless it does a tremendous amount of processing). Some Perl to native executable experiments have been done, but I have never managed to write anything in Perl that did not run almost instantaneously in the interpreter, so I have never investigated them. See Chapter 7 for a way to supplement Perl programs with routines written in C++.
3. But, yes, later shells such as the Korn shell and the C shell *do* have built-in math. Their use is a catch-22 in many respects, since the Bourne shell is the only standard UNIX shell. (The C shell is also fairly standard, but I have never heard a single good thing about its use as a shell programming language from people who should know.) If you use the Korn shell, you have to face the kinds of portability and availability problems you avoid using Perl. The Korn shell is standard with some UNIX flavors (like AIX), but not all; it is proprietary (not open source), so you will encounter "clean room" public-domain implementations on some systems; it exists in two major versions, '88 and '93, so you will have to be aware of the lowest common denominator, etc.
4. The only exception I can think of is Visual Basic.
5. The reverse is also true: Non-C syntax languages have enjoyed limited acceptance. VBScript lost big to JavaScript, IBM's revamped REXX for Windows is almost unused, Delphi is one of the greatest development tools of the 1990s "except it uses Pascal," etc. Switching languages that have different syntax is hard on the brain, especially if you have to do it often.
6. Randal Schwartz, upon reading an early version of the book, commented that awk just calls them "arrays," but the book *sed and awk* explicitly uses the term "associative array."

7. "Atomic" as in indivisible, not "atomic" in the nuclear explosion sense.
8. I mean *many.*
9. IBM originally called them fixed disks when they were first introduced. The name "hard disk" stuck, for whatever reason, and that is the name we use today. Fixed disk is rarely used but still appears in such vestigial places as the FDISK command (DOS and Windows 9x's fixed disk partition utility). The name "fixed disk" was a quantum leap in user-friendliness for IBM, however, since IBM originally called hard drives by the jargon-derived name "DASD" (Direct Access Storage Device, pronounced "DAZZ-de").
10. Other than paranoia, I mean. Some of the conspiracy theories you read on Perl newsgroups concerning ActiveState, Windows, Bill Gates, and so forth would be funny if you didn't think the people proposing them were serious. If you're paranoid that something horrible has been added to precompiled Perl binaries, then you're on your own.
11. If you are unfamiliar with PL/I, its main feature is a huge, cumbersome runtime library that has everything but the kitchen sink. Unlike later, modular languages, however, PL/I was a third-generation language like COBOL that had to have the entire runtime library linked with your program to do anything useful.
12. Contributing to this habit of writing in Perl 4-ish code is the fact that I missed out on the introduction of Perl 5. I graduated from college right when Perl 5 was coming out, and I spent a year or two grinding out C code all but cut off from the Internet. I did write Perl, but used the last release of the Perl 4 interpreter. After I got back into hardcore Perl programming a couple of years ago, I had to play catch-up by learning Perl 5.
13. I assume that computer types are familiar enough with Tolkien's *Lord of the Rings* to understand this analogy.
14. Granted, the software's being free makes it appealing to a certain segment of all computer users (educational institutions, people learning on their own, very small companies), but for professionals in the industry (defined as people who can spend other people's money on tools) lack of cost alone is simply not enough. For example, I have free rein to purchase development tools if they'll write software that my company can sell, and still I use Perl because it is better.

Chapter 1

A Cron Clone for Win32

One aspect of Windows that administrators will quickly notice is that it lacks a standard job scheduler, and there are too many nonstandard ones. Cron is the job scheduler that runs on all UNIX systems, and since it is standard there it would be nice to have on Windows, too. Cron runs in the background, noninteractively starting jobs at specified times. The first thing I want to build with Perl is a clone of the UNIX cron command.

What Is Cron and Why Use It?

Cron is a UNIX process that lurks in the background and carries out jobs as specified in `crontab` files. Cron is built into UNIX. Most administrators allow users to have their own `crontab` files (although you don't have to). You can use Cron to start jobs periodically, say, every night or once a week.

If you come to Windows from a UNIX background, the lack of a job scheduling facility quickly will become obvious to you. True, Windows NT does have a kind of job scheduler that comes with it out of the box. It's the `at` command. If you know UNIX, though, it would never enter your mind that the `at` command could schedule jobs to run periodically. (On UNIX, the `at` command runs a command once and only once.)

Windows 95 and 98 are worse because, between the two operating systems, there's no standard built-in job scheduler. Thanks to this lack of a built-in scheduler tool, a profusion of third-party schedulers exist. Some of these third-party schedulers provide general scheduling capabilities and some exist only to allow a product that needs a scheduler to have one (an example is Symantec's Norton AntiVirus). Microsoft has muddied the waters by adding a job scheduler to the Plus! pack for Windows 95 (the System Agent) and then integrating the System Agent scheduler into the Windows 98 Scheduled Tasks feature. If you're developing an application that needs to be scheduled, you can have 100% certainty that a Windows 9x machine *might* have a scheduler for it, but you may not know which scheduler or whether the scheduler will provide the capabilities you need.

When I began to automate procedures such as nightly backups on my network, I realized I wanted a Cron for Windows. Since I could not find any PC equivalent to Cron, I wrote this Cron clone.

The Cron clone has the following advantages.

- Since the Cron clone is 100% pure Perl code, you can include the Cron clone in

other Perl programs, which makes it easy to distribute software with scheduling capabilities.

- Since the `crontab` file is plain text, it is easy to generate new entries and to modify existing entries noninteractively. If you need to schedule jobs from other programs, this Cron clone might therefore be a good choice. (I don't know if the Scheduled Tasks folder can be modified noninteractively or not.)

A final advantage of recreating system administration tools such as Cron in Perl (as reviewer Sergei Listvin pointed out in his review of this book) is that Perl is more portable than C code. Perl code tends to be more easily ported to different platforms than C code because Perl is a higher level language.

The `crontab` File

The entries of the `crontab` file tell Cron what to do and when to do it. I discuss the highlights of `crontab`,[1] but nothing is better than reading a real UNIX manual page to get the feel. If you don't have a UNIX machine (or something similar, like a Linux box), one of the many books on UNIX systems administration should be a good substitute.

`crontab` files are what you would expect from the UNIX environment. They are cryptic and strange to the uninitiated, but they make sense once you grow accustomed to them.

A `crontab` file has one command per line. Each command begins with a string of numbers telling when to execute the command. The list of numbers is so flexible that you can set up any type of recurring job at any time interval you want. With this flexibility comes complexity, though. The hardest part of configuring a `crontab` file is getting the various fields in the right order. The trick is that the numbers on the line are position-dependent, and nothing other than your memory or a manual page reminds you which number means what unit of time. I always add a comment at the beginning of a `crontab` file with the names of the fields.
Each line is in the format:

```
min hour monthday month weekday command
```

The rules for the `crontab` file are:

- A line may begin with a # character, in which case it is a comment.
- Blank lines are ignored.
- Each field on a line is separated by a space.
- A number can either appear as a number (ex: 3)…

- …a comma separated list of numbers (ex: 3,4,5)…
- …or a * which means all possible values.

Each field is in the following range:

- minute: 0-59
- hour: 0-23
- day of the month:1-31
- month of the year: 1-12
- day of the week: 0-6 with 0=Sunday

Notice that the month field goes from 1-12.[2] If you're not a Perl programmer, a range of 1-12 for months may seem totally normal. If you know Perl, however, this range of 1-12 may seem confusing since the Perl `localtime()` function returns an array of which one member is the month of the year in the range 0-11. (The 0-11 format comes from the `struct tm` in C, which comes from the UNIX way of keeping time. The pervasive influence of UNIX on the C standard library caused the `struct tm` to be codified into the ANSI standard. The rationale for 0..11 range is that it can be used as an index into an array of names of months.) Since the UNIXism of having a 0..11 month range is so pervasive, if you know UNIX or Perl you'd expect the `crontab` range to be 0..11 too. But it is not, which could trip you up.

Of course, if this is your first exposure to Cron and you don't have a firm background in Perl or C, you will find the 1-12 range perfectly reasonable and normal. Don't be surprised later, however, when you learn that the Perl `localtime()` function returns a number from 0 to 11.

Example `crontab` File

To give you a feel for the `crontab` syntax, I provide two example `crontab` files. The first example is the file I use to launch nightly backups (discussed elsewhere in this book) along with a different, weekly backup:

```
# min hour monthday month weekday command
0 23 * * * start e:\backup\nightly.bat
45 23 * * 5 start e:\backup\weekly.bat
```

The first line is the comment I always include telling what the `crontab` fields are. The second line runs the batch file `nightly.bat` every day at 23:00 hours, or 11:00 p.m. The second line runs the batch file `weekly.bat` every Friday (day 5) at 23:45 hours, or 11:45 p.m. The batch files themselves handle all of the details of running the right programs to handle the specific jobs.

The following `crontab` file launches the calculator every minute:

```
 # min hour monthday month weekday command
* * * * * calc.exe
```

Launching Cron Automatically

Cron is most useful if it runs automatically when the system boots. UNIX Cron starts automatically at system boot, and the same behavior is desirable in Windows. How you configure Windows to run Cron automatically depends on what flavor of Windows you are using. The procedure is different on Windows NT and 9x.

On an NT machine that acts as a server (that is, users do not log on to it regularly), I recommend setting up one account that logs on automatically. Details are available in the Microsoft Knowledge Base article Q97597. An automatic logon account is the only way I have found to successfully run Cron on a server. Otherwise, if the machine reboots, it just sits there until someone manually logs on. Once you've set up the computer to automatically log on, create an entry to run `Cron` in the Startup folder for

the account that automatically logs on.

On a Windows 9x machine, to get a program to run when the machine boots but before anyone logs on, you have to perform brain surgery[3] on the Registry. I finally wrote a program called Run Control 98, which does the surgery for me and reduces the possibility of errors. Run Control 98 is a program that allows you to safely edit the information about which programs run automatically on your Win 9x system. Using it allows you to avoid directly changing the registry. You can find Run Control 98 on the CD that came with this book. I use this program to start programs that need to run before someone logs on, like Cron and pcANYWHERE, on Windows 9x machines. I've made this program available to everyone because I find it to be extremely useful.

To set up Cron to run automatically before anyone logs on, start Run Control 98 and switch to the Run Services tab (see Figure 1.1). Click Add…, and add the full path to the Perl interpreter, the full path to the `cron.pl` program, and the full path to your `crontab` file. You need to specify the full path, because there is no way to set a working directory for the program.

Figure 1.1 Run Control 98 Run Services Tab

The "Command to be run for item" field is so long that it does not fit into the edit box. It says:

```
c:\perl\bin\perl.exe
d:\scott\work5\autobook\cdrom\cron.pl
d:\scott\work5\autobook\cdrom\crontab
```

Click okay. and the Run Services list will appear as shown in Figure 1.2.

Figure 1.2 Run Control 98 Run Services List

(The other entries, other than Cron itself, will vary from system to system.)

The Code

Listing 1.1 shows the Perl code for the Cron clone. Some notes on the code follow the listing.

Listing 1.1 Cron Clone

```
```

```
#########################################################

cron -- by Scott McMahan
#
# Usage: cron <crontab file>
# Defaults to file named "crontab" in current directory
#########################################################



use strict;


#########################################################


# If you're not familiar with crontab files, they have
lines
# with the following:
#
# min hour monthday month weekday command
#
#  - Lines beginning with '#' are comments and are
ignored.
#  - The numeric entries can be separated by commas --
e.g. 1,2,3
#  - Ranges for each are as follows:
#           minute (0-59),
#           hour (0-23),
#           day of the month (1-31),
#           month of the year (1-12),
#           day of the week (0-6 with 0=Sunday).
#########################################################



#########################################################
```

```
# configuration section
##########################################################
################

# Note there are two levels of logging: normal logging,
controlled by
# the logfile variable, logs all commands executed and
when they're
# executed. The message facility prints messages about
everything that
# the program does, and is sent to the screen unless the
msgfile
# variable is set. Use the msgfile only in emergencies,
as voluminous
# output is generated (so much so that leaving msgfile
active all night
# could exhaust available disk space on all but the most
capacious
# systems) -- its primary purpose is for emergency
debugging.
# Due to the peculiar construction of the log functions,
the log and
# msg files can be the same file. This may or may not be
good.

my $logfile = "cronlog.txt";
my $msgfile = ""; # assign this only in emergency

# end of configuration

############################################################
#############
# in_csl searches for an element in a comma separated
list
############################################################
#############
```

```
sub in_csl {
    my ($what, $csl) = @_;
    CronMsg("Processing CSL");
    my @a = split(/,/, $csl);

    my $x;

    for $x (@a) {
                CronMsg("is $what equal to item $x?");
                if ($what eq $x) {
                      return 1;
                  }
    }
    return 0;
}

#############################################################
#############
# Range support -- suggested by Robin Berjon
#############################################################
#############
## Note: I have not had time to do range support or
include it in
## the cron processing logic. This subroutine should get
you started
## if you want to add it!
##
## sub in_rng {
##   my ($what, $myrng) = @_;
##   CronMsg("Processing range");
##   my ($begin_rng, $end_rng) = split(/-/, $myrng);
##   return 1 if ( ($what >= $begin_rng) && ($what <=
$end_rng) );
##   return 0;
## }
```

```
#############################################################
#############
# main program
#############################################################



my $crontab;

if (defined $ARGV[0]) {
    CronAct("using $ARGV[0] as crontab file\n");
    $crontab = $ARGV[0];
}
else {
    CronAct("using default file crontab\n");
    $crontab = "crontab";
}

while (1) {

    open(F, "$crontab") or die "Can't open crontab; file
$crontab: $!\n";
    my $line = 0;

    while (<F>) {
                $line++;

                if (/^$/) {
                      CronMsg("blank line $line");
                      next;
                }

                if (/^#/) {
                      CronMsg("comment on line $line");
```

```
                    next;
                }


        my ($sec, $min, $hour, $mday, $mon, $year,
            $wday, $yday, $isdst) = localtime(time);



                $mon++;
                 my $date =
sprintf("%0.2s/%0.2s/%0.4s", $mon, $mday, $year+1900);
                 my $time=
sprintf("%02.2s:%02.2s:%02.2s", $hour, $min, $sec);


                my ($tmin, $thour, $tmday, $tmon,
            $twday, $tcommand) = split(/ +/, $_, 6);



                CronMsg("it is now $time on $date;
wday=$wday");
                CronMsg(
                   "should we do $thour:$tmin on
$tmon/$tmday/--, wday=$twday?");


                my $do_it = 0; # assume don't do it
until proven otherwise


                # do it -- this month?
                if ( ($tmon eq "*") || ($mon == $tmon)
|| &in_csl($mon, $tmon)) {
                     $do_it = 1;
                     CronMsg("the month is valid");
                }
                else {
                     $do_it = 0;
                     CronMsg("cron: the month is
invalid");
                }
```

```
                # do it -- this day of the month?
                if ( $do_it && ( ($tmday eq "*")
                                          || ($mday ==
$tmday) || &in_csl($mday, $tmday)) ) {
                     $do_it = 1;
                     CronMsg("the day of month is
valid");
                }
                else {
                     $do_it = 0;
                     CronMsg("the day of month is
invalid");
                }

                # do it -- this day of the week?
                if ( $do_it && ( ($twday eq "*")
                                          || ($wday ==
$twday) || &in_csl($wday, $twday)) ) {
                    $do_it = 1;
                    CronMsg("the day of week is valid");
                }
               else {
                     $do_it = 0;
                     CronMsg("the day of week is
invalid");
                }

                # do it -- this hour?
                if ( $do_it && ( ($thour eq "*") ||
                                          ($hour ==
$thour)|| &in_csl($hour, $thour) ) ) {
                    $do_it = 1;
                    CronMsg("the hour is valid");
                }
                else {
```

```
                    $do_it = 0;
                    CronMsg("the hour is invalid");
               }

               # do it -- this minute?
               if ( $do_it && ( ($tmin eq "*") ||
                                   ($min == $tmin)
|| &in_csl($min, $tmin) ) ) {
                    $do_it = 1;
                    CronMsg("the min is valid");
               }
               else {
                    $do_it = 0;
                    CronMsg("the minute is invalid");
               }

               if ($do_it) {
                    chop $tcommand;
                    CronAct("executing command
<$tcommand>");
                    system ("start $tcommand");
               }
    }

    close(F);
    CronMsg("***-----***");

    # The message is to reassure users who might
encounter the window,
    # not know what it is, and wonder if it should be
closed.

    print "

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

```
**************************
     * DO NOT CLOSE THIS WINDOW *     * DO NOT CLOSE THIS
WINDOW *
     **************************
**************************


     The cron program runs in the background and kicks
off jobs
     at certain times. It is an essential part of the
system and
     required for keeping the network functioning
properly.


     **************************
**************************
     * DO NOT CLOSE THIS WINDOW *     * DO NOT CLOSE THIS
WINDOW *
     **************************
**************************


";


    sleep(60);
}


exit;



############################################################
#############
# Log activity
############################################################
#############


sub CronAct {
```

```
  my ($sec, $min, $hour, $mday, $mon, $year, $wday,
$yday, $isdst) =
    localtime(time);
  $mon++;
  my $date = sprintf("%0.2s/%0.2s/%0.4s", $mon, $mday,
$year+1900);
  my $time= sprintf("%02.2s:%02.2s:%02.2s", $hour, $min,
$sec);


  print "cron [$date $time]: @_\n";


  # since we're appending the log, always open it only as
little
  # as necessary, so if we crash the info will be there
  open(LOGFILE, ">>$logfile") or return;
  print LOGFILE "cron [$date $time]: @_\n";
  close(LOGFILE);
}


#############################################################
#############
# routine to log messages
# logs to screen unless $msgfile is set to a filename
#############################################################
#############


sub CronMsg {


  my ($sec, $min, $hour, $mday, $mon, $year, $wday,
$yday, $isdst) =
    localtime(time);
  $mon++;
  my $date = sprintf("%0.2s/%0.2s/%0.4s", $mon, $mday,
$year+1900);
  my $time= sprintf("%02.2s:%02.2s:%02.2s", $hour, $min,
$sec);
```

```
  print "cron*[$date $time]: @_\n";

  return unless $msgfile;

  # since we're appending the log, always open it only as
little
  # as necessary, so if we crash the info will be there
  open(LOGFILE, ">>$logfile") or return;
  print LOGFILE "cron*[$date $time]: @_\n";
  close(LOGFILE);

}
```

Code Notes

This code does three main things:

1. Reads and parses the `crontab` file
2. Executes jobs if there are any
3. Pauses one minute

This code decides if the right time has come to execute a command by checking the time from the largest unit of time to the smallest. If all the time settings in the `crontab` file match the actual time on the system clock, the command is executed. Perhaps a Perl guru could design a one-liner using a regular expression for this time-checking operation, but for the rest of us, this longer and more expressive code does the trick.

> Remember that Perl is designed to allow you to code as you think. Just because some Perl guru can turn any solution into a jam-packed and obtuse one-liner doesn't mean you have to. The Perl philosophy is that there's more than one way to do it, so you should write Perl code however you feel most conformable. If you feel comfortable writing an algorithmic solution to a problem that could also be solved using regular expressions, then write the algorithm. Perl coding tends to degenerate into "one-liner one-uppsmanship" from time to time, but you don't have to play that game if you don't want to.

Using the decision flag `$do_it`, each unit of time that can be specified is checked from the biggest to smallest unit of time. If any unit in the current time doesn't match the

`crontab` entry's time, then `$do_it` is cleared and the line is skipped.

The subroutine `in_csl()` looks to see if a number is in a comma-separated list of numbers.

One interesting thing to note about this program is the `CronMsg()` subroutine. I left this subroutine in to demonstrate how I create a self-debugging program. In a non-interactive program, it is often desirable to log everything that is going on. But what do you do with the output? You can't send it to the screen, because no one is watching.[4] You can either log it to a file or e-mail it to someone. Instead of using print statements, I write a function such as `CronMsg()` and call it anywhere I wish to print debugging output.

DO NOT CLOSE THIS WINDOW: The message about not closing the window gets printed every time Cron rests. This message lets casual passers-by know the window is important and doing something. For some people, the natural reaction is to close anything they do not understand. With this warning, hopefully, no one will inadvertently close Cron.

I made some design decisions that you may or may not agree with. If you don't like them, you can always rewrite the code:

1. I sleep one minute between intervals of processing the `crontab` file. Since Cron itself has a granularity down to the minute, I felt this was a safe way to not hog processor time. I also do not ever use a `crontab` file that is so big that processing it would take longer than a minute.
2. I reread the `crontab` file every time it is processed. For my needs, this does not impose a performance penalty. If you are experiencing performance problems, you may want to rewrite the code to load the file into an array and scan the array instead. I originally wrote the code this way to make debugging during development easier, since this method allowed me to modify the `crontab` file as the program runs. I never changed the code because I found this design did not affect the performance noticeably.

Having the `Cron` window either on the screen or minimized bothers some people. I believe one of the most frequently-asked questions concerning Perl, and anything else that runs in a console window, is how to hide the window. My Grab Bag 98 utilities (available at `http://cyberreviews.skwc.com`) include a program called `hide` that will hide, and show, any window on the system. `hide` also will print out a list of all the windows on the system. The `hide` command can run as part of a batch file.

There's more than one way to do it! Case in point: to me, the most natural way to express the thought "if there is a command line argument" is the code, `if` (defined $ARGV[0]). Schwartz suggested the alternative `if (@ARGV)`, which is a much more pure Perlish way of saying the same thing. The latter example relies

on your remembering that an array, when evaluated in a scalar context, returns the number of items in the array, and that an `if` statement's conditional *is* a scalar context. I'd never remember that in a million years, hence the code I wrote explicitly tests to see if `$ARGV[0]` is defined. The first part, that an array returns the number of items in a scalar context, is on p. 49 of *Programming Perl,* 2nd ed. You must deduce that the if statement *is* a scalar context by reading on page 22 that an if statement's conditional evaluates to a truth condition, and then realize that page 21 contains the statement that: "Truth in Perl is always evaluated in a scalar context." Schwartz is correct, and his way is more compact, but to apply it requires a language lawyer's familiarity with Perl (or memorizing it as an idiomatic expression).

The code uses the pattern `/^$/` to skip blank lines. This pattern could be made more complex by expanding the definition of a "blank" line. If you want to allow a blank line to contain whitespace, use the pattern `/^\s*$/`.

Schwartz pointed out when he read an early manuscript that `localtime()` defaults to time. This observation is true of Perl, but not of C.[5] I do so much C programming that an intelligent, time-saving default is surprising. I tend not to use defaults like this, because it makes going back to C just that much more painful.

The original Perl interpreter introduced the `chop` function, which removes the last character (regardless of what it is) from a string. Since `chop` was used most often to remove a trailing new line from a string, Perl 5 added `chomp`, which handles this special case. I use `chop` in this code-–because the `Cron` interpreter is very old, one of the oldest pieces of code I have. For the use here, `chomp` is more appropriate. `chomp` is a Perl 5 invention that I was not familiar with back when I wrote this code. `Cron` dates back to the time when I first began using Perl 5 on Windows machines, back when I knew Perl 4 much better than Perl 5. I am not alone in this affectation of defaulting back to the dialect of a language I learned first.

As an aside, I've found using `chomp` has one very important function in Perl programs: any program that uses `chomp` will fail to compile under a Perl 4 interpreter. This feature can be very useful as an early warning system if you port your code to computers that could potentially have either both Perl 4 and Perl 5 installed or only Perl 4 installed. If you're running under either the wrong interpreter or an incompatible version, a call to `chomp` guarantees your program will not get past the compile stage. This might be such an important safeguard that you'll want to put a spurious call to `chomp` in your program if you wouldn't ordinarily use it. `chop`, of course, works the same in both Perl 4 and Perl 5.

Don’t Mess with Imperfection

My Cron code is not perfect. I wrote the code to meet a specific need (running backups automatically, as described in the next chapter) and stopped when it did what I needed. The code as it stands has been extremely useful to me over the past few years, and I have used it in many other places. The Cron clone is not the first piece of software I’ve written that has been extended to uses far beyond its original intent.[6]

Ranges are absent simply because I never use them, so I have never taken the time to code them. UNIX Cron allows you to specify a range using the syntax first-last. (Using "3,4,5,6" in my Cron is okay, "3-6" is not.)

The Cron clone doesn’t explicitly handle daylight savings time. I start all my stuff before midnight, and I’ve never scheduled anything in the overlap time. I’m not even sure what would happen to a program that used the time during an automatic daylight savings time change.

I do not support embedded newlines in commands to allow running multiple commands from entries. I recommend that you alternatively write a batch file that does what you want in the order you want it done, and run the batch file from Cron. I typically find myself writing a batch file anyway, to set up environment variables and the `PATH`. The embedded newline support is a UNIXism that doesn’t translate to Windows very well.

I do not support one `crontab` file per user, and I do not support the allow/deny mechanism of UNIX for the same reason: NT is a single-user-at-a-time system. I don’t have multiple users using the same NT computer, and I have total control over the `crontab` file. If someone needed a job run, I could add it to the single `crontab` file.

Notes

1. The element "-tab" in a UNIX filename, especially a program’s configuration file, is an abbreviation for "table." This element is also seen in source code from time to time. This element is almost never used in Windows for filenames or in code. Any time you see it, you can be sure a UNIX programmer was involved somehow.
2. There seems to be some variation among all the various UNIX flavors. The UNIX flavors I am most familiar with (AIX and Solaris) use 1-12. My Linux box’s manual page for Cron says the range is 0-12! Perhaps anticipating the need to return to an occasional, collective thirteenth month in the calendar?
3. This is an imperfect analogy, since the human brain is better documented than the Registry. The parallels between the old `WIN.INI/SYSTEM.INI` and the new Registry are extremely interesting: although they have different interfaces, and the Registry was supposed to simplify system configuration, it seems as if nothing really changed: system information is still an impenetrable, poorly documented, and fragile mass. I do not believe this is intentional; I think that the sheer amount and complexity of information involved in Windows system (and application) settings is too much for traditional mechanisms. Any traditional organization of data will quickly disintegrate into the same impenetrable mass of fragile data.
4. In this program, I do print `CronMsg()` output to the screen if a logfile is not specified, but not really for the purpose of reading it. The reason Cron spews its `CronMsg()` output to the screen is that I

frequently run it on Windows 9x machines, and the constant stream of messages is to reassure me that the Cron process is still running and has not locked up, and to show the machine itself has not locked up.

5. C has a `localtime()` function, too, but you'd better believe there's no default!

6. A true story: I once wrote an u-u-ugly program for a one-shot situation with no intent to reuse it. It did a hex dump of a BLOB (binary large object) field in a database. Within the week, the beta site of the program that created the BLOB called and asked if there was any way to see the data in the BLOB field. I had the program and sent it to them. The program is still part of the product to this day. The point of the story: no code exists that doesn't have the potential for being used far beyond its original purposes and for taking on a life of its own.

Chapter 2

Sending Mail the SMTP Way

My automated tasks send me e-mail because when one does something, I like to know about it. In this chapter, I will show you how to add SMTP-based e-mail to your programs.

MAPI Doesn't Cut It

The goal of this chapter is to develop a Perl subroutine that will send a piece of e-mail. Mainly, this subroutine will be used to send noninteractive e-mail, a curiously forgotten and neglected part of Windows. I do not believe (from the difficulty I've had doing it) that noninteractive e-mail was ever considered by whoever designed Windows' e-mail support. The design seems to center on interactive use. The untapped potential for noninteractive e-mail is huge. When an automated system performs a task behind the scenes, e-mail is the most natural way for it to report what transpired. E-mail is certainly convenient, since it is dumped directly into your inbox rather than written to a log file in a forgotten directory.

Let me explain why I do not use MAPI (via Automation) and instead use an SMTP-based solution. I have learned through my experiments that Microsoft's MAPI, the Messaging API, is one of the biggest losses in Win32 history. A little background on MAPI reveals why it is such a loss, and how I have learned from my own experience not to use it.

Like much of Microsoft's COM technology that you meet in this book, MAPI is a misnomer. MAPI is not an API in the traditional sense of function calls such as the Win32 API; it is instead a mind-boggling series of COM interfaces. MAPI is also not an implementation. MAPI is an architectural standard to be implemented. Any given e-mail system can implement MAPI. The key problem with MAPI is that no two implementations are quite the same. As I show you soon, this discrepancy is even true of Microsoft implementations.

MAPI is a mire of mind-boggling complexity even for experienced developers. A mere glance at a book such as *Inside MAPI* is enough to make you reconsider supporting e-mail in an application at all. Whence came this complexity? For an answer, go back to the dawn of MAPI, a time before the Internet had met Windows. Back before 1994, the PC world was a place of many different online services, all with incompatible, proprietary front ends. All these services had e-mail of a sort (indeed, e-mail was their main attraction), but each service was separate and isolated in its own

little world. Brand X's e-mail was in one proprietary format, and Brand Y's in another. Microsoft's answer to this was the *universal inbox*, which would allow any online service to write a mail transport agent and any e-mail client to access the mail downloaded and stored locally on a PC.

Consider a stadium, built in a team's glory days to seat a hundred thousand spectators, after the team's fortunes have waned and the team barely draws a few hundred people. The stadium stands lifeless and empty. The same thing happened to MAPI when the explosion of proprietary e-mail systems became an implosion as everyone began using Internet standards for e-mail. After 1994, the Internet standards for exchanging and addressing e-mail began to totally replace all proprietary standards, and the proprietary systems had to begin supporting Internet standards to talk to one another. MAPI was a grand design, for its time, but the time passed before MAPI became widely used. The completeness of MAPI for plugging in support for new mail subsystems, however, gave Microsoft a valuable tool for rapidly integrating Internet mail into their applications. For that reason alone, MAPI was an important strategic move for Microsoft. But with the multitude of online e-mail formats reduced to basically SMTP and POP3,[1] MAPI itself became somewhat redundant and even Microsoft jettisoned it for simpler, faster "Internet mail only" versions of e-mail client programs.

As it happened, the mail engineers at Microsoft were just about the only people who implemented MAPI. I have tried to make a list of all the Microsoft E-mail clients, but can't guarantee this list is complete.

- The original Windows Messaging that came with Windows 95
- The Exchange 4.0 client
- The Exchange 5.0 client
- Internet Mail and News
- Outlook 97
- Outlook Express[2]
- Outlook 98

No two of these clients support MAPI the same way, particularly when it comes to the black hole of using Automation to send e-mail noninteractively.

For Perl use, there is a `MAPI.Session` function that could be called through the `Win32::OLE` interface. It should be avoided. I'll give two examples:

1. I once spent considerable time writing a command line e-mail program (in C++) using the Common Messaging Calls (CMC), a simplified wrapper around MAPI itself. I finished the program only to find it was useless in CGI programs (which is a serious reason to send non-interactive e-mail, after all) because the user under

which CGI programs ran did not have permission to use MAPI.

2. I once tried to use `MAPI.Session` in a Perl program, and when I tried to send a message, a dialog box kept popping up with the mail message in it. I had to manually press Send to send the message. I tried this on a different MAPI version, a different e-mail client, and I had a different result: nothing happened at all. Depending on the e-mail client installed, `MAPI.Session` will exhibit different behaviors.

`MAPI.Session` is not supported by Microsoft, and I do not think it ever was. Microsoft does not acknowledge that it exists and does not document it. It is possible to reverse engineer some of the methods and their parameters based on the full-strength C++ MAPI itself, but this approach is not advisable.

At the time of this book's writing, Microsoft was working on a successor to `MAPI.Session`, a new e-mail-handling object model also based on Automation. Time will tell if this new model will be successful or not, or better than its predecessor. Unless Microsoft fixes the underlying problem that the e-mail client itself provides an implementation of the Automation object and MAPI subsystem, I suspect its efforts to devise a new e-mail-handling object model will fail, since the resulting component will be vulnerable to the same problem `MAPI.Session` had: different clients provide different behaviors. The only solution is to build MAPI into Windows itself and make e-mail clients shells that use the MAPI object model.

This new Automation system for mail should debut in Office 2000's Outlook 2000, which happens to be the first Outlook implementation to support Visual Basic for Applications. Many people wonder why all the Office applications (even the paperclip) support the real, full-strength VBA, while Outlook is stuck with the comparatively wimpy VBScript. The answer is that MAPI is entangled with the broken, useless `MAPI.Session` object. VBA is based on using an application's document object model to manipulate the documents the application is responsible for creating (thus `Word.Application`, `Excel.Application`, etc.). Outlook, on the other hand, doesn't have a document object model for VBA to operate on. Hence, VBA isn't supported in Outlook because Outlook has no object model to support VBA. Its "document" is simply the MAPI messaging system. Outlook is a shell around MAPI. But since MAPI has no Automation interface, Outlook has to use the direct C++ and COM API for MAPI. Once Microsoft develops an object model for MAPI that works, Outlook will get VBA. The fact that Outlook has no VBA support is the most telling evidence of how broken `MAPI.Session` really is.

> If something is so broken even Microsoft won't use it, that ought to tell you something. Microsoft began achieving a reputation in the computer industry in the late 1990s for having less-than-adequate quality control, although I am happy that the organization has rededicated itself at the end of 1998 to improving the

situation. It is also interesting to speculate that Outlook 2000 is going to be the third version of Outlook. Conventional wisdom in the computer field says, and historical trends show, that Microsoft usually perfects software the third time. If you count Outlook 97, 98, and now 2000, the e-mail client is at this magic third version. The world has reason to expect that Outlook 2000 will be considerably improved and very stable. I hope these improvements also include a stable Automation interface!

So if MAPI for non-interactive e-mail is a total loss, what can you use instead?

A Quick Look at SMTP

The Simple Mail Transfer Protocol (SMTP) is the Internet standard way of sending e-mail. The SMTP protocol is simple. (SMTP is, however, somewhat picky about the syntax, since it is not designed to be used by people but implemented by computer programs.) A typical e-mail message can be sent using four commands. If you have Internet mail protocols on your network, you almost certainly have an SMTP server available. Using SMTP allows you to do an end run around Windows' MAPI baggage.

RFC 821, "SIMPLE MAIL TRANSFER PROTOCOL," by Jonathan B. Postel, defines the Simple Mail Transfer Protocol. This protocol has remained stable since August 1982, which proves how well-tested and reliable this protocol has proven to be. (By contrast, Microsoft didn't even have any e-mail products then.) The name RFC originally meant "Request for Comment," but RFCs have become documents in which Internet standards are defined. The documents themselves are somewhat hard to read, since they are written in a formal standardeze language meant to be precise first and readable second. (It is not uncommon to see LEX/YACC-suitable syntax diagrams in RFCs.) If you do much e-mail programming, you will want to have a copy of RFC 821 available as a reference.

RFC 821 begins by saying: "The objective of Simple Mail Transfer Protocol (SMTP) is to transfer mail reliably and efficiently. … SMTP is independent of the particular transmission subsystem." One reason SMTP has become the standard mail protocol on the Internet is because it *is* simple, and it can be implemented easily[3] by a wide variety of mail servers. Typically, when you think SMTP, you think UNIX (because historically UNIX machines were the only machines that implemented SMTP), but Exchange 5.0 also has SMTP capabilities.

It is possible to telnet to port 25 of your SMTP server and manually send mail using the raw SMTP protocol that your e-mail client generally hides from you. You may wish to do this as you follow the discussion about how the SMTP protocol works. Enter SMTP commands and see the results.

I begin with an SMTP transaction and dissect how it works. In this example, a user named `someguy` is sending mail to `scott`. The `mailhost` referred to in the messages is the SMTP server. SMTP uses a numbering scheme for its messages that is similar to the scheme used by the HTTP protocol.

```
220 mailhost ESMTP Sendmail 8.8.7/8.8.7; Mon, 30 Nov 1998
18:01:58 0500
HELO somehost
250 mailhost Hello localhost [127.0.0.1], pleased to meet
you
MAIL FROM: someguy
250 someguy... Sender ok
RCPT TO: scott
250 scott... Recipient ok
DATA
354 Enter mail, end with "." on a line by itself
From: someguy
To: scott
Subject: just a note!

This is just a note, explaining how SMTP works.


.
250 SAA26077 Message accepted for delivery
QUIT
221 mailhost closing connection
```

The mail message `scott` receives looks like:

```
Return-Path: <someguy>
Date: Mon, 30 Nov 1998 18:02:20 -0500
From: someguy@mailhost
To: scott@mailhost
Subject: just a note!
```

```
This is just a note, explaining how SMTP works.
```

When you connect to the SMTP server, it prints out a message identifying itself.

```
220 mailhost ESMTP Sendmail 8.8.7/8.8.7; Mon, 30 Nov 1998
18:01:58 0500
```

The first thing you should do is identify yourself:

```
HELO somehost
250 mailhost Hello localhost [127.0.0.1], pleased to meet
you
```

Note that the format is `HELO myhost`, *not* `HELLO remotehost`. The argument to `HELO` is the host sending mail, not the host to which you connect. Many programs that send mail via SMTP get this backwards. SMTP's `HELO` is backwards from "hello" in English conversation. In English, you say hello to someone else. SMTP's HELO identifies *you* to the other party in the conversation.

To begin sending mail, use MAIL FROM: followed by the address of the sender:

```
MAIL FROM: someguy
250 someguy... Sender ok
```

Next, specify who gets the message using `RCPT TO`: followed by the recipient's address. The recipient's address can be a standard Internet e-mail address, or it can be an address relative to the SMTP host to which you connect.[4]

```
RCPT TO: scott
250 scott... Recipient ok
```

To send the message itself, use the `DATA` command. You then supply the text of the e-mail message. Terminate the message text with a dot on a line by itself. You must insure the body of your message does not contain any lines with just a dot. Replace any single-dot lines with a line containing two dots. Otherwise, your message will terminate prematurely. (The most effective thing to do if you're writing an SMTP transaction client over which you'll have full control of the messages being sent is just to never use a line with a dot on it. Whether you handle this condition depends on how motivated you are and who will be using the software.)

```
DATA
354 Enter mail, end with "." on a line by itself
```

```
From: someguy
To: scott
Subject: just a note!

This is just a note, explaining how SMTP works.


.
250 SAA26077 Message accepted for delivery
```

It is important for you to realize that the `MAIL FROM:` and `RCPT TO:` commands *are not part of the e-mail message*. Just as the envelope for a business letter is a wrapper with the recipient's (and your) address, the SMTP transaction itself (with the `MAIL FROM:` and `RCPT TO:` commands) is an envelope around the e-mail message.

Your e-mail message text should comply with RFC 822, "STANDARD FOR THE FORMAT OF ARPA INTERNET TEXT MESSAGES." RFC 822 is the standard for what an e-mail message itself looks like. For most quick and dirty utilities, over which you have full control of the message, omitting the e-mail message header is okay. I have never seen an implementation of SMTP that would not make up an RFC 822 compliant header for you.[5] A minimal header is generated if no header is present. If you are sending mail to a real Internet mail host and not just your own machine on the network, or if you are processing someone else's input and e-mailing it, you should put a full, formal header on the message.

When you are finished, use QUIT to quit.

```
QUIT
221 mailhost closing connection
```

At this point, the mail message has been sent.

Before QUIT, you may send as many different e-mail messages as you want to by repeating the MAIL/RCPT/DATA process.

A Function to Send Mail with SMTP

I have encapsulated the preceding discussion into a Perl subroutine called `smtpmail()`, which will send an e-mail message for you. `smtpmail()` is shown in Listing 2.1.

The `smtpmail()` function has been designed to be called from other programs. You can't run it by itself. Therefore, I have placed the function in a module called `smtpmail.pm`. Other programs can use this module and call the function (as shown in

the demo). The demo program introduced in the next section and the backup program presented in the next chapter use the function.

The `smtpmail()` function is called with three parameters: the address to which the message will be sent, the subject of the message, and an array containing the lines of the message itself.

```
use Smtpmail;
Smtpmail::smtpmail($to, $subj, @msg);
```

The mail looks as if it is *from* whatever e-mail address has been defined in the variable `$mf` in the `smtpmail` module itself. I am assuming non-interactive use of this module by a background process, so the mail will not actually have to be replied to. The from address, therefore, is unimportant and can be set to something like `webmaster` or `nobody`.

My function has been tested both with UNIX sendmail and Exchange Server 5.0, so it works with both varieties of SMTP servers. (Most Perl SMTP code that I've seen won't work with Exchange, because the Exchange server requires a much slower feed of the information and the program spits it out faster than the server can accept it.) The cryptic line `$| = 1` is also essential—it changes the buffering on the socket that is opened to communicate with the e-mail server.

My `smtpmail()` function assumes you are sending to known e-mail addresses on the local intranet and transmitting messages you have full control over (i.e., messages you created yourself) and do not need much error checking. I do not check for SMTP error codes. Instead, I just pour the mail into the SMTP server and assume everything is okay. I have used this function with my nightly backup for two years, and I have never had a single error condition occur. If the SMTP server is unable to process the mail, something is going to be so wrong that I easily notice the error manifesting itself in some other way. If you are going to use this function to send mail to addresses that are not known to be good in advance, or if you plan to send messages you will not have control over, you may wish to add some error checking to this function.

> My `smtpmail()` long predates the availability of the Perl module `Net::SMTP` on the Win32 platform. Now that `Net::SMTP` is available with Perl 5.005, you may wish to investigate it to see if it meets your needs. I do not know if `Net::SMTP` will work with Exchange servers, since I no longer use Exchange and don't have a way of testing.

Listing 2.1 Demo of the `smtpmail` Function

smtpmail()

```
package Smtpmail;
use Exporter;
@EXPORT = qw(smtpmail);

#############################################################
############
# smtpmail
#
# This function has been specifically developed to work
with
# Microsoft Exchange Server 5.0's Internet Mail Service.
It is based
# on the typical UNIX-based Perl code for an SMTP
transaction.
#
# You pass this subroutine the e-mail address to send the
message
# to, the subject of the message, and an array of strings
that
# makes up the body of the message. The common idiom to
build the
# array of strings is to push a string onto the array
every time
# you would print out (or in addition to printing out)
information
# to the console.
#
# This subroutine must be configured to work on your
intranet before
# you use it!
#############################################################
```

```
############


$mf =  "webmaster";
$whoami =  "localhost";
$remote  = 'mail';


use Socket;


sub smtpmail {
  my ($to, $subj, @msg) = @_;


  my ($port,   # the mail port
      $iaddr, $paddr, $proto, $line); # these vars used
internally


  $port    = 25;
  $iaddr   = inet_aton($remote)               || die "no
host: $remote";
  $paddr   = sockaddr_in($port, $iaddr);
  $proto   = getprotobyname('tcp');
  select(SOCK);
  $| = 1;
  socket(SOCK, PF_INET, SOCK_STREAM, $proto)  || die
"socket: $!";
  connect(SOCK, $paddr)    || die "connect: $!";


  # The sleeps in this transaction are necessary in order
to let
  # the Exchange server work with your


  print SOCK "HELO $whoami\n";
  sleep(5);
  print SOCK "MAIL FROM: $mf\n";
  sleep(5);
  print SOCK "RCPT TO: $to\r\n";
```

```
  sleep(5);
  print SOCK "DATA\n";
  print SOCK "From: $fm\n";
  print SOCK "Subject: $subj\n";
  print SOCK "To: $to\n";
  print SOCK "\n";
  print SOCK @msg;
  print SOCK "\n.\n";
  sleep(5);
  print SOCK "QUIT\n";

  sleep(5);

  close (SOCK)            || die "close: $!";
}
```

Listing 2.2 is a brief demo program that shows the idiom of building a message by pushing lines onto an array. Some notes on the code follow the listing.

Listing 2.2 smtpdemo.pl

]

smtpdemo.pl

```
############################################################
############
# smtpdemo.pl is just a test driver to excercise the
# smtpmail.pm module. From this little program, you can
see
# how easy it would be to extend this to e-mail a file
# automatically, or even e-mail the text of the command
line
# arguments to someone. The possibilities are endless.
############################################################
############
```

```
use Smtpmail;


$to = "scott\@softbase.com";
$subj = "Automating Windows With Perl";


push @msg, "Dear reader,\n\n";
push @msg, "Hello and welcome to Automating Windows With
Perl.\n";
push @msg, "This book will make you realize the power of
Perl\n";
push @msg, "in a Windows environment, and unleash your
own\n";
push @msg, "creative juices to make your network run
better and\n";
push @msg, "with less human intervention\n";
push @msg, "\n";
push @msg, "Sincerely,\n\n";
push @msg, "Scott McMahan (author)\n";


Smtpmail::smtpmail($to, $subj, @msg);
```

Code Notes

Module names need to be capitalized, hence this module is `Smtpmail` and not `smtpmail`. Just as identifiers beginning with an underscore and a capital letter are reserved for the implementation in the C language, so are module names beginning with a lowercase letter reserved for the implementation in Perl. User modules ought to begin with a capital letter, so there will never be any namespace collision between user and implementation modules.

The message array `@msg` can be built in many different ways. I use the example of one push statement per line. This is typically the best example for a program that builds the array over a period of time in widely separated statements. Here, all the push statements are bunched together, but in another program they could easily be interspersed throughout many lines of code.

Upon reading my code, Randall Schwartz suggested:

```
push @msg,
  "line one\n",
```

```
  "line two\n",
  "line three\n";
```

I have also frequently used this syntax inspired by lists:

```
@msg = ("line one\n", "line two\n", "line three\n");
```

Many other ways exist to add new items to an array. There's more than one way to do it, and the best method is the one that most naturally expresses the context of the solution.

To be fully, 100% standards compliant, all the lines in `@msg` should end with `\r\n`. I use `\n`, which is the lazy way to do it, first because I have never seen an SMTP implementation that couldn't figure out this syntax, and second because I do most of my coding in C, and never remember to use `\r\n`. The two reasons are closely related: I would be more careful if there were any reason to be, but since my SMTP servers will accept what I write, I just ignore it. The line between adhering to standards and "winging it" (with the idea that the standard is what the implementation does) is a personal decision you have to make for yourself. I tend to try to reduce the cognitive dissonance I incur when switching between programming languages (which I do frequently). Most of my Perl code, for example, looks like C code because I switch between the languages so much.

> E-mail addresses like `"scott\@softbase.com"` are a problem until you get used to them. An `@` in a double-quoted string must be escaped in Perl v5 (although not in Perl v4). The program fails to compile if you do not escape the `@`. You can, of course, use single quotes to avoid this trap: `'scott@softbase.com'` works fine. Sometimes, though, you'll find it desirable to do variable substitution and have an e-mail address in the same string.

Notes

1. POP3, the third version of the Post Office Protocol (don't confuse this mail POP with the ISP Point Of Presence acronym) is the opposite of SMTP: where SMTP allows you to send mail, POP3 allows you to download it.
2. Despite the name, "Outlook Express" has nothing to do with the Outlook 9x family.
3. I mean, of course, easy for e-mail. Nothing about e-mail is particularly easy, but some things are easier than others. For example, if you are new to Perl and UNIX and come from a Windows background, be gald you have never had to edit a `sendmail.cf` file. This UNIX mail configuration is not easy.
4. Because of widespread spamming, most Internet mail hosts place severe restrictions on who can use their SMTP server to send e-mail. Usually only hosts in their domain are allowed to send mail. If you

connect to an SMTP server from another domain, you will likely get an error saying the host will not relay mail.

5. Unfortunately, here is a situation where spamming is causing some newer SMTP implementations to reject a message that is not RFC-822 compliant. My code is very old and assumes an SMTP server from the kinder, gentler days. If your SMTP server does not accept mail that is not RFC-822 compliant, you will have to include the full header. I am assuming you are using SMTP on an intranet where you have control over the SMTP server and know if this is the case. For general use on the Internet with arbitrary SMTP servers, you should always use full standard-compliant messages.

Chapter 3

Automated Nightly Backups

Administrators do not need to be told the need for a good backup system. I won't tell you about the need; in this chapter, I will develop a system that uses Perl along with the freely available InfoZip software to automate nightly backups for a small workgroup.

Backing Up

This chapter presents the backup strategy I use as the network administrator for my company. I work for a small company with a single file server and no more than 350MB of backup data per night. This backup data includes our entire range of mission-critical business data from documents to phone messages. My strategy assumes that I'll never have more than 1GB of data to back up at night. Having more data than a single tape of the largest capacity you have available (1GB in my case) is a good indication of the need to use an industrial-strength backup solution.

If your needs are greater, you will probably want to look into a commercial backup and restore system that will provide industrial-strength backups. Windows machines, both NT servers and computers on which Windows 9x is used as a file server, are typically used in small- to medium-sized companies. Many readers are in the same position I am in: a big, commercial backup package is overkill (too expensive and complex for the amount of data), but a personal-sized backup package is insufficient. From my experience, the low end "personal" backup packages don't have enough features for unattended nightly backups. (They also suffer from proprietary backup formats, which I will discuss soon.)

My strategy is to use two familiar tools, Perl and ZIP, to create a system for doing automated backups.

Why Use Perl and ZIP?

Backup programs are a dime a dozen, so I must begin by explaining why I prefer to use ZIP and Perl to do my backups.

1. Proprietary backup formats generally can't be read on other operating systems and computer architectures. Generally, you must restore the backup from the same platform you backed it up on. This situation isn't always what you want. Likewise, it is hard or impossible to transport backups in proprietary formats to other platforms, say for archival or long-term storage.

(Although now, with the arrival of tapes that look like hard drives to the operating system, proprietary backup tools can create files. They do not have to format the media themselves. Unfortunately, the file formats are still proprietary.)

2. I have not evaluated many backup programs (particularly recently), but I don't know of any backup programs that send e-mail on completion. I want a full report in my mailbox every morning about what went on last night while I was asleep.
3. The restore utilities of backup programs are usually a pain to use. I've never used one I enjoyed. ZIP archives, on the other hand, are so widespread that you can pick and choose among utility programs for unzipping. ZipMagic 98, WinZip, the venerable but still indispensable Norton File Manager, and others provide easy alternatives for restores, particularly partial restores.

What makes a Perl and ZIP solution so attractive is the recent widespread adoption of tapes that look like disk drives, such as the Iomega Jazz (and Zip)[1] drive. Even though the Jazz drive is a tape, it looks to the computer like a disk, which means it is easy to create ZIP files on a Jazz drive.

A Nightly Backup Program

The key to my backup program is simply that it is part of a larger system. All night, when everyone has gone home, things are happening automatically on the network. My SQL Server database creates a backup copy at a certain time. The phone system backs itself up about the same time. And so on. Finally, near midnight, my Cron program (see Chapter 1) kicks off my backup program, the final phase. It backs everything up to my Jazz drive. When the backup is finished, the backup program e-mails me a report of everything that happened.

I prefer e-mail reports because I like the interrupt-driven nature of e-mail. If a program creates a log file somewhere, with all the distractions and deadlines swirling around me everyday, there is almost zero chance I will ever read that log file. It could be weeks or months before I notice a problem. Instead, I like to have the report dumped into my mailbox every morning. I can then read the report as part of a routine.

I use the InfoZip command line tool to create the ZIP backup file on the Jazz drive. I use ZIP because it is a familiar tool that I use all the time. One problem with custom backup software is that the restore procedure is unfamiliar. Even if you do periodic drills to practice restore procedures (and, honestly, who has time to do periodic drills?), you're still going to be rusty when it comes time to do a real restore under normal circumstances simply because you don't use the software that much.

ZIP, on the other hand, is a file format I deal with daily. I have many good tools to use on ZIP files, both for packing and unpacking. I am familiar with these tools because I use them all the time.

I am not in the business of endorsing products, but two products I recommend are InfoZip's command line ZIP utilities (which are free and available on many platforms, as well as included on this book's CD-ROM) and PowerDesk from Mijenix, a tool for working with ZIP files. PowerDesk is a commercial package. For shrink-wrapped software, PowerDesk is amazingly robust and reliable. (I am profoundly skeptical of and cynical about shrink-wrap software.) But PowerDesk and its close relative ZipMagic (an amazing shell extension for Windows NT and 9x that makes ZIP archives look just like regular folders) work and don't crash. Many people swear by WinZip, a product I have never been able to warm up to.

Another advantage of ZIP is that it is easy to test ZIP files, especially noninteractively. The InfoZip command line program has the `-t` switch for testing. You can easily create, for example, Cron jobs that automatically test the integrity of backup files and e-mail you the results.

ZIP files are also your protection against obsolescence. ZIP is here to stay. ZIP has become the de facto universal compressed archive format of computing. By using ZIP, you protect yourself from unreadable file formats. Even if a future OS upgrade (or replacement) can't manage ZIP files, it won't be long before a ZIP clone appears for the new OS. (The InfoZip source code is freely available and extremely portable.)

Code

The Perl nightly backup program is shown in Listing 3.1. Notes on the code follow the listing.

Listing 3.1 Nightly Backup

```
############################################################
############
# Nightly backup
############################################################
############
# by Scott McMahan, Sept. 1997
############################################################
############


use strict;


use smtpmail;
```

```
use Socket;


#################################################################
#############
# CONFIGURATION SECTION
#################################################################
#############


my $maxgen = 4;


my @dirs = ("c:\\some directory",
            "d:\\other dir",
            "e:\\etc");


my $genfile = "e:\\backup\\generation.txt";


my $src = "e:\\";
my $dest = "h:\\";


my $zipcmd = "zip";
my $ziptestcmd = "unzip";
my $zipadd = "-9r";
my $ziptest = "-tq";


### END OF CONFIGURATION SECTION
#####################################


#################################################################
#############
# Setup stuff
#################################################################
#############


my @msg;
```

```
# mon = 0..11 and wday = 0..6
my ($sec, $min, $hour, $mday, $mon, $year, $wday, $yday,
$isdst) =
  localtime(time);
$mon++;


my $date = sprintf("%0.2s/%0.2s/%0.4s", $mon, $mday,
$year+1900);
my $time= sprintf("%02.2s:%02.2s:%02.2s", $hour, $min,
$sec);


my $day=$mday;


msg("started at $date $time\n");


############################################################
############
# What generation is next?
############################################################
############


msg("Getting next generation\n");


# $x is the current generation, if you get the joke...


open(G, $genfile) || die "Can't open generation file
$genfile, $!\n";
my $x = <G>; # only one line in file
close(G);


msg("This is generation: $x");


############################################################
############
```

```
# Create an 8.3 filename for this file
############################################################
############

msg("Creating 8.3 filename\n");

msg("Month is $mon\n");
msg("Day is $day\n");
msg("Year is $year\n");

my $bfile = sprintf("$src%2.2d%2.2d%2.2d%2.2d.zip", $x,
$mon, $day, $year);

msg("New filename is: $bfile\n");

############################################################
############
# Zip to the real drive
############################################################
############

msg("Creating zip file\n");

my $files = join(" ", @dirs);

my $z = "$zipcmd $zipadd $bfile $files";

msg("command: $z\n");

open(ZIPADD, "$z|") || die "Couldn't execute $z, $!\n";

while(<ZIPADD>) {
  msg(" zip: " . $_);
}
```

```
close(ZIPADD);


############################################################
############
# Test created zip file for integrity
############################################################
############


msg("Testing zip file integrity\n");


$files = join(" ", @dirs);


my $zt = "$ziptestcmd $ziptest $bfile";


msg("command: $zt\n");


open(ZIPT, "$zt|") || die "Couldn't execute $zt, $!\n";


while(<ZIPT>) {
  msg(" zip: " . $_);
}


close(ZIPT);


############################################################
############
# Delete oldest generation
############################################################
############


msg("Deleting oldest generation\n");


# actually, the oldest is the one we'll do next!
```

```
my $oldgen;

if ($x == $maxgen) {
  $oldgen = sprintf "%2.2d", 1;
}
else {
  $oldgen = sprintf "%2.2d", $x + 1;
}

msg("Old generation is $oldgen\n");

my $delcmd = "command /c del $dest$oldgen*.zip";

msg("Command is: $delcmd\n");

my $rc = system($delcmd);
#unlink("$dest$oldgen*.zip");
sleep(5);

msg("Return code from delete is $rc\n");

############################################################
############
# Move over old file
############################################################
############

msg("Moving over new file\n");

my $movecmd = "move $bfile $dest";
msg("Command is: $movecmd\n");
```

```
$rc = system($movecmd);
sleep(5);


msg("Return code from move is $rc\n");


############################################################
############
# Increment generation in file
############################################################
############


msg("Incrementing generation & saving\n");


if ($x == $maxgen) {
  $x = 1;
}
else {
  $x++;
}


open (G, ">$genfile");
print G "$x\n";
close(G);


msg("New generation is $x\n");


############################################################
############
# Send report
############################################################
############


# mon = 0..11 and wday = 0..6
($sec, $min, $hour, $mday, $mon, $year, $wday, $yday,
$isdst) =
```

```
  localtime(time);
$mon++;


$date = sprintf("%0.2s/%0.2s/%0.4s", $mon, $mday,
$year+1900);
$time= sprintf("%02.2s:%02.2s:%02.2s", $hour, $min,
$sec);


msg("ended at $date $time\n");


sleep(10);


print "*** mailing...\n";


smtpmail::smtpmail("scott\@softbase.com", "Backup
Report", @msg);


print "*** ... done mailing\n";


############################################################
############
# Post backup step (reboot system, etc)
############################################################
############


exit;


############################################################
############
# Subroutines
############################################################
############


# msg prints a message, and stores it for later too
```

```
sub msg {

  my $line;

  $line++;

  for (@_) {
    my $out = sprintf "%4.4d sbbackup: $_", $line;
    #print "This is the line:\n\t$out\n";
    push @msg, $out;
    print $out;
  }
}

sub newsect {

  push @msg, "-" x 70 . "\n";

  print @msg, "-" x 70 . "\n";

}
```

Code Notes

You need to set up the variables at the beginning of the program to reflect what you want backed up and where you want it backed up. The array `@dirs` is a list of all of the directories I want backed up. (Although I typically back up entire directories, it is also possible that individual files could be backed up as well.) Note that I use the `\` path separator, which in Perl must be `\\` (in a double-quoted string). I use this separator because these filenames will be passed to external programs that expect the backslash path separator.

This program can keep multiple generations of the backup. The program deletes the oldest generation and adds the new generation it creates. `$genfile` is where I store the current generation between runs.

`$src` and `$dest` require a little explanation. `$src` is the drive on which I create the ZIP file. I do not like to create the ZIP file directly on the removable media because it is

so slow compared to a real disk drive. I have plenty of room for the ZIP file to reside temporarily on the real disk. `$dest` is the removable media on which I want to put the ZIP file.

Another reason for creating the ZIP file on a real drive first is that the removable media can fill up. If the ZIP file can't be moved to the removable media, it will still be on the disk drive and you'll still have a usable backup. Occasionally, I have experienced the situation where some extraordinary, unplanned increase in the total size of the files occurs before I can adjust the number of generations kept, and the backup will not successfully copy into the removable media.

The filename is in the form GGMMDDYY.ZIP, where:

- GG = the generation number
- MMDDYY = month/date/year

I intentionally kept the filename to eight bytes with a 3-byte extension because the 8.3 format is the most universal format for transferring files between machines.[2]

The most interesting aspect of this backup program is the fact that I use the `msg()` function to keep track of all the activities this program performs. As lines are added using the `msg()` function, they are pushed onto an array. At the end of the program, I e-mail the contents of this array to myself so I have a full report of every file that has been backed up. (The `msg()` function also prints its message to the console, because a DOS window opens and stays open. When I come in, before I even check my e-mail, I can look to see the return codes for the steps of the backup job.)

Notice I also log the output of the ZIP command itself:

```
open(ZIPADD, "$z|") || die "Couldn't execute $z, $!\n";

while(<ZIPADD>) {
  msg(" zip: " . $_);
}
```

With this code, I have a log of every single file that has been backed up.

Note that I say:

```
$files = join(" ", @dirs);
```

When he read the manuscript, Randall Schwartz suggested the alternative:

```
$files  = "@dirs"; # use interpolation to add spaces
```

This solution surprised me, because in the case of printing an array (e.g., `print @dirs`), spaces are *not* added.

The spaces are, however, in an interpolation context, but an array used by itself is a scalar (e.g., my `@x = qw{ 1, 2 }; $tx = @x; print $tx;`—the printed value is 2, the number of items in the array). This kind of situation makes my head hurt, and I prefer the simplicity and self-documenting nature of the join.[3]

Schwartz also commented that my code to increment the next generation (`if ($x == $maxgen`), etc.) could be replaced by a strange construction involving a `%` operator.[4] I take his word—I am the most non-mathematical person currently working in the computer field, by a wide margin. I need every step spelled out in great detail to follow a mathematical calculation. If you prefer his solution, replace my if statement with

```
$oldgen = $x % $maxgen + 1;
```

I can't explain exactly why I used a shell call to move the file when I could have used Perl's `rename()` function. I suppose I used a shell call because every other operation I was doing involved shell commands, and I was just thinking shell commands. If the only tool you have is a hammer, everything looks like a nail. You could, of course, use the built-in `rename()` function to accomplish what my call to the shell does.

Where to Get InfoZip

All of the examples in this book use the `unzip` and `zip` commands from InfoZip. Other ZIPs may be used, but the command line syntax will differ. I prefer InfoZip because it is free software, and it is available on many different platforms. All platforms use the same code base and have the same command line syntax.

InfoZip can be downloaded from:
`http://www.cdrom.com/pub/infozip/Info-Zip.html`

It is also included on the CD-ROM that accompanies this book.

Notes

1. Unfortunately, the Iomega Zip drive is a terminology clash. It has nothing whatsoever to do with the ZIP files we're talking about here.
2. There is no Y2K bug in the 2-digit year, because the date is for informational purposes only. The generation number, not the date, controls how the file is aged.
3. The Perl array join should not be confused with an SQL join. Same word, different concepts.
4. Not a `printf %`, either. The %, in the context of being an operator, is a "modulo." I am not qualified to explain what it does and instead refer you to a mathematics book. I have been writing programs of one sort or another since 1984, and I can't recall ever needing to use this operator in a program. Maybe I've been missing out on something.

Chapter 4

Automating Office 97

Microsoft ought to sell Office to programmers as the “Reusable Automation Components Pack.” Few programmers realize that Office is essentially a suite of reusable business components for an extremely low price. You can integrate these Office components into your own custom applications. Office 97's reusable components let you write versatile and powerful business applications in a fraction of the time you would otherwise have to spend. Office 97 is also one of the purest examples of Microsoft's Automation philosophy. This chapter examines how you can add the power of Office 97's components to your Perl programs through Automation.

This chapter assumes you have Microsoft Office 97. The same concepts apply to other applications that provide Automation interfaces. (Lotus SmartSuite, for example, fully supports Automation.)

As this book was going to press, Microsoft released Office 2000. I expect everything in this book will work in that environment, but I have not been able to test with it.

> The closest thing to a book advocating Office 97 as a collection of reusable components is the *Microsoft Office 97 Developer’s Handbook*. For a discussion of Office as a source of reusable code, see the Microsoft Office 97 Developer's Handbook.

A Brief History of Microsoft

To understand what Automation is, you must first understand where it came from. I'll begin with a brief look at some recent Microsoft history. This background will be particularly useful to UNIX programmers coming to Windows, since UNIX has no parallel to Automation.

Windows today has a long legacy that dates back to the first personal computers that ran the primitive DOS operating system, which barely did enough to qualify as an OS. DOS could load a program from a file into memory, run it, and do I/O, but that was about it (other than a few things like setting the clock). Because DOS was so limited, and only one program could be run at a time, applications assumed they were the only program running on the computer. For performance reasons, the applications often had to replace parts of DOS. Since all hardware (back then) was standardized around the IBM PC's architecture and even non-IBM PCs followed the same design,

programs could assume certain hardware was present and that it had certain capabilities. Therefore, the application could bypass the OS and use the standardized hardware directly. Multitasking came much later, and the mentality of applications developers slowly adjusted to the fact that multiple programs could run at the same time. The concept of a program running noninteractively was the hardest for Windows developers to realize, since they had single-tasking DOS roots and were used to programming an interactive GUI. UNIX was the exact opposite in almost every way. UNIX applications assumed the hardware they ran on was totally unknown, and UNIX applications were often shipped in source format to be recompiled on radically different hardware. These applications depended on the operating system to insulate them from the hardware. UNIX applications also assumed they would be running with other applications at the same time. UNIX also has strong roots in the pre-PC teletype days, where most applications were designed to be run noninteractively. In UNIX, the need for something like Automation didn't exist.

In the "good old days" of DOS, the computer could run only one application at a time, so the ability for one program to run another program noninteractively was irrelevant. As programs became more powerful and users became more sophisticated, users wanted ways to automate repetitive tasks by storing up several operations and executing them as a single task. This single task that incorporated several operations was generally called a *macro*, and, with macros, the applications became primitively programmable (although these macros did little more than record keystrokes and play them back).

As macros became widely used, power users and programmers wanted more than just straight-line execution. They wanted the macro to be capable of making decisions and looping just like programming languages. Thus was born what became known as the *macro language*. Applications had grafted onto them, in a backwards fashion, the control structures of procedural programming languages. This support generally became entangled with the keystroke macro facility already present. These languages reflected the structured programming paradigm almost completely, allowing looping and decision making.

Macro languages tended to fall into two camps: those that were based on the BASIC language and those that were not. The former relied on BASIC’s strengths. BASIC was attractive to the application developer faced with choosing a macro language. It offered an easy, interpreted language to implement, and it was easy to extend BASIC to work with the application itself. For end users, BASIC was familiar and easy to use (or, if not familiar, easy to learn). The languages of the second camp usually had some specific reason for not using BASIC, normally because of an existing macro language or users who would rather have a different macro language. Lotus’s 1-2-3 macro language (in the DOS days, I mean, since 1-2-3 later had the BASIC-like LotusScript grafted onto it) evolved out of the program’s `@` functions and had a delightfully eclectic, weird syntax. Programmers' editors tended to use C-like macro

languages, which were more familiar to application developers who worked in C.

The important point is that the early macro languages did almost nothing but provide the same options and commands as the user interface. The user interface was the application. As DOS faded and Windows took over, this paradigm did not substantially change. (In large part, undoubtedly, because Windows applications were often ports of the DOS core application engine. At any rate, the same DOS programmers usually brought their skills with applications development forward for Windows programming.) Programmers didn't generally differentiate between the application and its user interface. The application and the interface were one and the same: what it looked like was what it did. Macro languages still were little more than BASIC with built-in functions that copied the menus, often with the exact same names.

As Windows grew more powerful and multitasking became accepted, this whole macro approach broke down. Applications had their own macro languages, and each was different. Even though most were based in some way on BASIC, the application macro languages had their own dialects. No two were the same.

The final straw that sent Microsoft looking for a better way was the fact that as multitasking took over Windows and people used multitasking extensively, the interprocess communications (IPC) method built into Windows showed how limited and unscalable it was.

Dynamic Data Exchange (DDE) was a protocol that enabled an application to contact another application and send a stream of bytes to it. (And I mean *was*: the technology is outdated and is only still in Windows to support legacy programs; no one should actually use it anymore.) The other application read the bytes and did something with them. The major problem with DDE is that the back-and-forth talk was totally undefined. As applications wanted to talk to each other noninteractively, they faced the difficulty of not knowing what another application wanted to hear. DDE communication formats were largely a black pit of undocumented functionality. No one used DDE much, and it primarily served as a way to tell Program Manager to create new groups and icons (one of the few well-documented procedures performed over DDE). No standards emerged.

Microsoft eventually scrapped DDE and started over again with the Component Object Model (COM). Microsoft created COM to remedy all the problems with DDE. They created Automation, which is built in terms of COM, to fill the glaring problem of how applications could run each other noninteractively and how scripting languages could run applications.

For some reason, Microsoft has made a mess of naming their technologies. Originally, OLE was Object Linking and Embedding, a technology that ran on top of the old DDE. Microsoft released OLE 2.0, which was the first version built on top of COM instead of DDE. Once this happened, Microsoft began renaming their technologies OLE (as in *OLE Automation* and *OLE Controls)*. This didn't last long, because it confused people terribly. The emphasis on OLE de-emphasized COM, which was the

technology Microsoft wanted people to be using! Microsoft then began emphasizing COM and renaming technologies built on top of COM. (OLE is still object linking and embedding, and it has the same functionality it has had since 1.0, but now OLE is written in terms of COM instead of DDE; ActiveX is the component technology built on controls written to COM standards; and Automation is the technology that lets clients run servers programmatically using COM for communications.) Compounding the confusion, books and articles both from Microsoft and third parties used different terminology at different times. Microsoft has not helped by constantly flooding developers with new names (DCOM, COM+, DNA). Overall, though, Microsoft has mostly just changed the terminology without changing the underlying technology. COM has not changed substantially since OLE 2.0 was introduced in the early 90s. Even the adoption of support for distributed computing (the D in DCOM) had been planned all along and introduced no major changes. COM was designed to use DCE's RPC (remote procedure call) standard from the beginning, and the non-D COM used LRPC, the lightweight RPC, which ran only on a single machine.

Today's Automation is the result of separating the functionality of an application from how other users, people, or programs access the functionality. The road from COM's introduction to today's Automation was long and bumpy, and it had a lot of dead ends. Microsoft went through a huge paradigm shift from the procedural programming model to the object-oriented programming model, and their core applications underwent many changes, culminating in the powerful paradigm of Automation and the document object model.

COM and Automation

COM is like plumbing or electrical wiring. It is the absolutely necessary infrastructure to support something else you want. You *want to* drink water or turn on the light, but to realize these wants, you require the water pipes and wires to be in place to supply you with water and electricity. The infrastructures by themselves are not particularly interesting or useful; only when these infrastructures function to help supply you with the things you want are they important. COM is the same way: as a programmer, you want components like ActiveX controls and Automation servers (and so on), and to realize these things you use COM's infrastructure. COM by itself is not very exciting, but the technologies which are based on it are.

COM itself is a standard way to communicate between different modules on a computer, or on different computers in the case of DCOM (*distributed* COM). COM is designed to be language-neutral, so any program written in any language can communicate with any other (as long as they both support COM), regardless of how different and incompatible these languages normally are. COM defines a standard for function calls and a standard way for data to be passed back and forth. Exactly how COM does this is not even of concern, because modern development tools are so

sophisticated that they hide the complexities of COM (such as IDL, ODL, and other strange-looking abbreviations). You as a programmer do not have to understand the underlying COM theory to use technologies built on COM any more than you need to understand electrical theory to turn on a light.

Automation is one of several technologies designed to use COM as its infrastructure. (Other technologies include OLE, ActiveX controls, MAPI, the Explorer shell, and so on.) Automation has two characteristics that make it ideal for Perl programming:

1. It is noninteractive. Automation has little overhead because of its noninteractive nature, which means you do not have GUIs and messaging to content with. Adding GUI support complicates things tremendously.
2. It is simple. Of all the various technologies built on COM, Automation is one of the easiest to learn and use. Automation is so simple that if you can write a DLL, you can write an Automation server.

I have done a considerable amount of mixed-language programming. I've had to call COBOL and C code from each other and from client languages like Delphi, Visual Basic, and others. (This is a true story: one system I analyzed in Excel! The best tool for the job is my motto.) One truth that has become obvious to me as I do this and gain more experience with it is that Automation is the best way to simplify and streamline mixed-language programming. In fact, calling code from one language from another is a black art requiring much trial and error. Its main drawback is that considerable time, which I assume you don't have to waste, must be devoted to figuring out how to pass incompatible data types back and forth and deciphering calling conventions. Automation puts the burden on the language implementation to support COM's way of doing both of these, relieving you of the burden of figuring out the details.

Automation consists of a client that invokes the services of a server. An Automation client is a program that uses the COM subsystem in Windows to request the services of an Automation server. Clients can be written in any language, as the example in this chapter shows, but typically they are in very high level languages such as Perl and Visual Basic.

In this case, the server will be an Automation DLL. These are the simplest, but the only, kind of Automation servers to create. Other types exist, for example, programs such as Word and Excel that are both regular executables and Automation servers. A "client" and a "server," in this case and in most cases, are relative, and the same software can be both a client and a server depending on what it is doing. Word, for example, is both an Automation server and an Automation client (via Visual Basic for Applications). It is possible to write an Automation server in almost any language (though I don't think you could do it in Perl at the time of this writing), but C++ is the most frequently used language.

One reason why Automation has not lived up to its potential is that, to date, languages have either allowed easy Automation access or been good for noninteractive programs. Visual Basic (including Visual Basic for Applications), Delphi, and other languages allow extremely easy access to Automation but make it hard (though not always impossible) to write noninteractive programs. C++ makes it easy to write noninteractive programs but makes Automation extremely difficult to use. Perl is unique in that it is designed for scripting noninteractive tasks and also has high-level, easy support for Automation client programming.

The ramifications of Automation have yet to fully appreciated, because there hasn't quite been enough time. Office 97 is a huge, powerful product with which people are just now coming to grips and getting familiar.

Automating Office 97 with Perl

With Office 97, Microsoft took a revolutionary step. The breakthrough was not simply Automation; rather, Automation fused with object-oriented programming. The vision Microsoft had of separating what an application did from how it did it took its final form when documents became objects. Microsoft calls this the document object model. An application's documents became objects with properties and methods, on which *any* Automation client could operate. Before this breakthrough, macro languages and even Automation servers were totally procedural. The document was inert. It just let the application operate on it. The object-oriented document is exactly the opposite. It knows what it can do and will do things to itself if you ask it (using Automation calls). The burden of knowing what a client can do has shifted from the client that wants to operate on the document to the document itself.

Office 97 upended the status quo by establishing three rules:

1. "Documents" are collections of different types of objects (the document object model), which are either objects or containers of objects. All of these objects expose methods that let you do stuff to them (for example, paragraph and section objects in Word, cell range objects in Excel, slide objects in PowerPoint). You tell a cell to format itself like currency, or you tell a paragraph to format itself with a certain style.

2. All objects exist as language-neutral Automation objects that can be operated upon by any Automation client, from a GUI shell to a non-interactive scripting language. The Automation client could be the supplied VBA, a Perl program, a Delphi program, or whatever. This is what Microsoft calls Automation.
3. The "application" the user sees is just a user interface shell that uses the functionality provided by the objects. The program you see on the screen is not the functionality itself, it's just a shell. The shell comes with a default configuration, but it is totally customizable. (This is the ideal, anyway: not all applications quite live up to this standard yet.)

Perl, in Windows, is the ultimate Automation client because it combines the extremely painless Visual Basic style of coding with Perl's strength as a scripting language for running jobs automatically. The best strategy for using Automation with Perl is to code as much of your automatic procedure in Visual Basic for Applications as is humanly possible and then call the VBA code from Perl. The interface between Perl and Automation is a gray area, and the best thing you can do is reduce the amount of Automation calls and the complexity of those calls from Perl.

If you don't know Visual Basic (VB), you should learn enough about it to get by. All Automation interfaces are invariably described in terms of VB (particularly the examples), and you must be able to read and understand VB to translate the Automation interface examples into Perl. VB is the de facto Automation client language. Users of Perl, Delphi, and other client languages must be able to translate the VB examples into the native language. VB should be trivial to learn if you know Perl and have ever been exposed to Basic.

All of your Perl programs need to `use Win32::OLE;` before they can be Automation clients. The `Win32::OLE` package is included with the Win32 port of Perl. (Perl must, of course, be properly installed so that Perl can find the `Win32::OLE` package. A normal install should set this capability up for you automatically.) Again, terminology confusion emerges from the OLE/COM naming—even though you are doing Automation client programming, the module is called `OLE`.

> In this book, I am describing Automation client programming using Perl 5.005. Older versions of the Win32 port had a different mechanism. If you see Automation client code in Perl that has `use OLE`, it is the older version. I recommend sticking with the new version and Perl 5.005. The only real difference between the two is the `use OLE` and `CreateObject` where I use `new` in this book.

You create an Automation object in Perl using `new`. An example:

```
$obj = new Win32::OLE 'ProductName.Application' or die
"Couldn't create new instance";
```

The call to new returns a reference to the object, which you can store in a normal scalar and use to refer to the object. The name of the object you create is its Automation name, which is usually in the form `ProductName.Application` (e.g., `Word.Application`). How do you know the name to use? You must encounter the name in documentation. (If you are comfortable with the Registry, you may encounter the names in `HKEY_CLASSES_ROOT`. You will still, however, need documentation on any given object's methods and properties. If you have no documentation at all, you could use Visual C++'s OLE/COM Object Viewer, a tool that shows a tremendous amount of information about COM objects.) Most of the time, the pattern `Program.Application` is followed, where `Program` is the name of the program that exposes the Automation interface.

The methods and properties of the object are stored as references in the original object. You use the Perl arrow operator to access them.

Once you have your first object, creating more objects is easy. If an object provides a method that creates another object, just assign the return value (which is an object reference) to a Perl scalar variable. Use the variable any time you need the object.

To call a method, just use a normal Perl method call. Example:

```
$word->Documents->Open($filename);
```

What makes Automation so easy to use is the fact that it is language-neutral and you don't have to perform data type conversions every time you call a method. One of the worst aspects of mixed-language programming is the fact that no two languages seem to use the same data type formats. With Automation, your client language, Perl or anything else, will either use OLE data types intrinsically or convert them for you when necessary.

Object properties are stored in a Perl hash. Idiomatically, you set properties with the construction

```
$object->{"property"} = $value;
```

An explanation of what's going on and why this works is beyond the scope of this book, but see a reference like *Programming Perl* for details. This idiom is equivalent to the VB code `Object.Property = Value`.

It is possible to use named parameters to Automation method calls in Perl, as you would in Visual Basic using the `:= operator`. The `Win32::OLE` module says: "Named parameters can be specified in a reference to a hash as the last parameter to a method call."

The idea of a reference to a hash is best described using an example:

```
$object->method( { NamedParam1 => $value1, NamedParam2 =>
$value2} );
```

Following is a skeleton for running a macro in Word 97. The code will work with little change in the other Office applications. Please note one thing: the macro must be a macro, not just a subroutine. It has to show up in the Tools>Macros>Macro... list.

```
# This code runs a macro in Word 97
#
# You will have to change the name of the DOC file to
# match the DOC file containing the code you want to run,
# and change the name of the macro you want to run.

use Win32::OLE;
$word = new Win32::OLE 'Word.Application'
  or die "Couldn't create new instance of Word App!";

# In noninteractive use, you probably want to comment out
# this line. For testing, it is good to have the app
visible.
$word->{Visible} = 1;

# Change the filename to the document you want to open.
$word->Documents->Open(
'd:\scott\work5\autobook\cdrom\HelloWorld.doc');

# Change "HelloWorld" to the name of the macro you want
# to run.
$word->Application->Run("HelloWorld");
```

```
# Like setting Visible to 1, you will want to comment
this line
# out for testing and restore it for production use. If
you
# are testing, you probably want to keep Word open to see
if
# the macro worked or not.
#$word->Quit();
```

The VBA code for saying “hello, world” is within the file HelloWorld.doc. You can use this piece of Perl code as a stub for your own macros. Once you're running VBA code, you can do anything the Office application can do. That's what makes my technique so powerful.

```
Sub HelloWorld()
'
' HelloWorld Macro
'
    ' Even in a noninteractive macro, it is possible to
add message
    ' boxes and show forms. You could leave Word hidden
and use
    ' its functionality to create an interactive
application.
    Beep
    x = MsgBox("Hello, Word 97 World!", vbInformation)

    ' You can easily add text to a document.
    Selection.TypeText Text:="Hello, Word 97 World!"
    ' There is no limit to how advanced a macro can get.
Here,
    ' Word creates a plain text file. Plain text is what
Perl
    ' works with best, so this technique is a good way to
get
    ' information out of a document into a format Perl
can read.
```

```
    Open "d:\hw.dat" For Output As #1
    Print #1, "I wrote this to the file"
    Close #1

End Sub
```

`Selection.Typetext` is Word 97's idiomatic command for adding text to a document. If no text is selected, the insertion point is an empty selection. This Word insertion point is kind of like the Emacs insertion point, which is between two characters.

> How do you figure out idioms like `Selection.TypeText` for text insertion? Don't overlook the power of the macro recorder! Remember that the macro recorder generates Visual Basic code. Any action you can perform interactively can be recorded. The beauty of the document object model is that there's no difference between interactive and noninteractive use. If you get stuck on how to express something in VBA, do what you want to do interactively while you record a macro and look at the VBA code the macro recorder generates.

Example 1: Automatically Printing A Word Document in Bulk

As a demonstration of how to automate Office 97 features using Perl, I show you a Perl script that prints a document noninteractively. The goal of this program is to create a batch process that can print a large document in bulk. The idea is to create a totally noninteractive process that can be scheduled to run late at night when the computers, network, and printer are idle. With a heavy-duty printer (such as one with a 2000 sheet tray), the job could run all night. Once the macro is developed, it can be stashed in `normal.dot` and used over and over.

> When you develop and debug a macro like this that can potentially cause a large number of copies of a document to be sent to the printer, you should disable actually printing to the printer and let documents queue up instead without printing. You can do this by going to Start/Settings/Printers, right click on the printer's icon, and either Pause Printing (for locally attached printers) or Work Offline (for networked printers). The documents will spool, but they will not actually print.[1] You can delete them from the spool without printing them.

The single most important file to Microsoft Word is `normal.dot`. On the surface,

`normal.dot` appears to be the default template used for blank documents. Over the years, though, `normal.dot` has become a magic file that serves as Word's own system registry for global settings. Many different types of important settings are saved in `normal.dot`, and much deep wizardry goes on with this file. If you are not backing up `normal.dot` regularly, start right now! `normal.dot` is located in the folder Templates under `C:\Program Files\Office`, or wherever you installed Office 97.

To begin the task of developing the bulk print application, create a Word macro that prints a document. Start Word, start the macro recorder, call the macro `PrintMeUp`, and choose File/Print. Print the current document. Now, stop macro recording.

Open the Visual Basic editor and drill down in the Project window to the `normal.dot` file. Find the module called `NewMacros`. The NewMacros module is where the macro recorder's work gets deposited. You should see something that looks a little bit like this:

```
Sub PrintMeUp()


' PrintMeUp Macro
' Macro recorded 04/27/98 by Scott McMahan
'
    Application.PrintOut FileName:="", _
    Range:=wdPrintAllDocument, Item:= _
    wdPrintDocumentContent, Copies:=1, _
    Pages:="", PageType:=wdPrintAllPages, _
    Collate:=True, Background:=True, PrintToFile:=False
End Sub
```

One thing you immediately learn about the macro recorder is that it is as verbose as possible, and it includes parameters that have defaults even if the defaults are chosen. This is a design decision someone made, and I imagine whichever team designed it had a long argument about it.

The preceding `PrintMeUp()` macro reveals a very important fact: printing is done through a method in the application object called `PrintOut`. I was initially surprised by this. What I expected, given Word's object model, was a method of the document object that would tell the document to print itself.

Usually, though, a design like this makes sense in a weird way if only you can figure it out. The biggest clue to understanding the print puzzle is the fact that Word uses the MFC library and its document/view model. The document is not responsible for printing itself in MFC. The view, which renders the document for a display device, is responsible for printing. The Application object, not the document, handles the view.

Two important parameters you need to give Word in order to print a document are the filename and the number of copies. Note that `FileName` and `Copies` are both parameters of the `PrintOut` function. The filename defaults to the current document if the `FileName` is blank, and the number of copies will be as many as the printer will allow. So make the subroutine look like this:

```
Sub PrintMeUp()
'
' PrintMeUp Macro
' Macro recorded 04/27/98 by Scott McMahan
'
    Application.PrintOut filename:="", _
           Range:=wdPrintAllDocument, _
           Item:=  wdPrintDocumentContent, _
           copies:=99, _
           Pages:="", PageType:=wdPrintAllPages, _
        Collate:=True, Background:=False,
PrintToFile:=False
End Sub
```

Notice that I changed `Background:=True` to `Background:=False`. I made this change because, as a background process, the need is to quit Word after printing as part of the batch operation. If Word is background printing, you can't quit. Worse, you get an interactive dialog asking if you want to cancel the print, which Perl can't respond to!

Note that I said copies should be the maximum number the printer will allow. Although you can include any fanciful large number for the number of copies, most printers have an upper limit to how many copies they'll print at a time. My Epson printer will allow no more than 99 copies (and its printer driver doesn't even report an error if you exceed the maximum; the printer just doesn't print anything at all if the requested number of copies is greater than 99). You will have to experiment with your printer to determine the maximum number of copies it will support.

Now that I have the macro, I'll use the `Application.Run` stub to call it:

```
use Win32::OLE;
$word = new Win32::OLE 'Word.Application'
  or die "Couldn't create new instance of Word App!";
$word->{Visible} = 0;
$word->Documents-
>Open('d:/scott/work5/autobook/cdrom/wordprint.doc');
```

```
$word->Application->Run("PrintMeUp");
$word->Quit();
```

This stub is the general model for how I approach using Office noninteractively. I create as much of the functionality in VBA as possible and call what I have created from Perl.

Example 2: A Boot Log in Excel

I've noticed a curious phenomenon with end users. Instead of reporting a problem with a PC, occasionally some users will be absolutely silent about it. They won't tell the network administrator that a problem exists. They will try to live with or work around the problem for as long as they can, until it gets to be so bad it impairs their ability to do their work. *Then* they'll tell the administrator. Of course, by then, the problem has gone on for days or weeks and gotten much worse than it would have been from the beginning. (And I won't even get into naïve users who create an even bigger mess by trying to "fix" the problem themselves.)

The most common way to solve any Windows problem, particularly with Windows 9x client machines, is to reboot. The way to track potential problems is to track the number of reboots. Any abnormal number of reboots should suggest trouble. Ideally, a user boots the machine once in the morning and then shuts it down at the end of the day. This is unrealistic, obviously. Buggy applications and glitches will cause users to reboot more than once during a day. By keeping historical data, you can get a feel for what a "normal" number of reboots for a given machine will be. The obvious way to record reboot data is in a spreadsheet. I originally approached this problem by trying to use Perl to insert the data into an Excel spreadsheet on each boot, but, in practice, this solution was too slow because it took an exceptionally long time to load Excel and then run the macro.

Instead, I created a plain-text log file using Perl that contains one record for each reboot. A line is added to the file each time the program runs.

```
############################################################
############
# bootlog.pl - keeps a record of when your machine boots
in Excel
############################################################
############


$default_bootfile = 'default.boot';
```

```
$default_reason = "Just because";


($sec, $min, $hour, $mday, $mon, $year, $wday, $yday,
$isdst) =
  localtime(time);
$mon++;
$year += 1900;
$date = sprintf("%0.2s/%0.2s/%0.4s", $mon, $mday, $year);
$time= sprintf("%02.2s:%02.2s:%02.2s", $hour, $min,
$sec);


$bootfile = $ARGV[0];
$bootfile = $default_bootfile unless $bootfile;


$reason = $ARGV[1];
$reason = $default_reason unless $reason;


open(BF, ">>$bootfile") or die "Can't open $bootfile:
$!";


print "Booting at $date $time for $reason\n";
print BF "$date|$time|$reason\n";


close(BF);
```

Note that in this code, the reason for the reboot can be specified as a command line parameter. I put in a reason parameter to allow for sophisticated reboot analysis. My Essential 97 shareware has a shutdown program that can be used from a batch file, allowing you to log different types of shutdowns. Fairly involved reboot scripts could be created if necessary.

Then, this VBA code in Excel gathers the data for any number of these log files and inserts it into a spreadsheet.

```
Sub gather()

Dim col As Integer

basedir = "d:\scott\work5\autobook\cdrom"

col = 2

Range("A1").Value = "File"
Range("B1").Value = "Boot Attempt"
Range("C1").Value = "Date"
Range("D1").Value = "Time"
Range("E1").Value = "Reason"
Range("A1", "E1").Font.Bold = True



MyPath = basedir & "*.boot"
MyName = Dir(MyPath, vbNormal)
Do While MyName <> ""
    MyLine = 0
    Open basedir & MyName For Input As #1
    Do While Not EOF(1)
        MyLine = MyLine + 1
        Line Input #1, wtext
        Range("A" & col).Value = MyName
        Range("B" & col).Value = MyLine

        firstbar = InStr(wtext, "|")
        Range("C" & col).Value = Left$(wtext, firstbar -
1)
        rest = Mid$(wtext, firstbar + 1, Len(wtext))
```

```
        secondbar = InStr(rest, "|")
        Range("D" & col).Value = Left$(rest, secondbar -
1)
        Range("E" & col).Value = Mid$(rest, secondbar +
1, Len(wtext))

        col = col + 1
    Loop

    Close #1

    MyName = Dir



Loop

End Sub
```

Note: You can't run the `bootlog.pl` program from `AUTOEXEC.BAT` under Windows 9x. At the time `AUTOEXEC.BAT` is processed, the Win32 subsystem hasn't started and the Perl interpreter can't run. You will need to put this program in the startup group or somewhere else, such as the `Run Services` entry in the Registry, that can be edited using Run Control 98. (If you want to run `bootlog.pl` from `AUTOEXEC.BAT` badly enough, you could get a DOS Perl clone.)

One final thing about this Excel code: since writing it for this book, I've adapted it several times to read *other* delimited files. Having this routine in my bag of tricks has proven valuable.

Final Notes on Automating Office 97

Trying to use Office's Automation from Perl can be extremely frustrating. Some sources of frustration are:

1. Automation is still a new technology, and the impact of it on Office is still being understood. Originally, Office's applications had their own self-contained macro languages, on top of which Automation was added. Experience quickly pointed out limitations, and the macro languages are being retrofitted to work better with Automation (or, in the case of Word Basic, are being scrapped entirely). Office is

also moving to the new object-oriented architecture that is radically different from previous versions. So it's no wonder that even Microsoft's official documentation is extremely confusing. Office is subject to rapid and fluid change in the Automation interface and object models from release to release.

2. Documentation on using Automation from Perl is almost nonexistent. A few books discuss it in passing, but 90% or more of the Automation example code in the world was written for Visual Basic. If you're going to program in Perl, you must be adept at reading Visual Basic code and translating it to Perl.
3. The Automation examples provided with the Win32 Perl distribution seem to be for older versions of Excel. Excel, of all the Office applications, has undergone radical mutations to its object model. Unlike Word, which made a clean break with the Word Basic past, Excel has been an experimental laboratory and a proving ground for Visual Basic for Applications and Automation over the years. From version to version, Excel changes in more subtle but radical ways than any other Office application.
4. It is impossible to debug Automation. All you can do is use the black-box approach: try something, see if it works, and if it doesn't, try something else based on your intuition about what you think went wrong. Obviously, this is unproductive and frustrating. Yet there are no real tools to help you: driving applications from Automation clients is still bleeding edge.

Many people get trapped in the poor documentation and trial-and-error approach to debugging and become so frustrated they give up. I hope that by applying the techniques in this chapter, which I have learned through trial and error, your experience with using Office from Perl will be smooth and easy.

Notes

1. Print spooling is the oldest surviving computer subsystem (that I know of) still in widespread use. It predates interactive terminals and multitasking by a wide margin. Originally, hard disks were added to mainframe computers to provide a cache for punched cards and printed output. The bottlenecks were the card readers and line printers, since the CPU could process jobs faster than these could input and output them. An entire job of innumerable punched cards would be read onto disk (older mainframe programmers still call lines of texts "cards") so the CPU could access them faster. Output would be spooled to disks to wait its turn on the line printer. Print spooling has endured as a useful idea long after many of its contemporaries have been replaced.

Chapter 5

Smoke Testing Your Application in Developer Studio

In this chapter, using Perl as an aid in software development is explored as automating project rebuilds is discussed. The *Microsoft Press Computer Dictionary, Third Edition*, says of a smoke test: "The testing of a piece of hardware after assembly or repairs by turning it on. The device fails the test if it produces smoke, explodes, or has some other unexpected violent or dramatic reaction, even if it appears to work." The concept of a smoke test has been borrowed from electronics and adapted to computer programming, where it is unquestionably useful. A software smoke test is the process of building an application and testing *it* to see if it "produces smoke" – that is, fails.

Using Developer Studio Noninteractively

In this chapter, I discuss using Perl to rebuild Developer Studio projects automatically for the purposes of smoke testing. Although Developer Studio is an integrated development environment, it can also run noninteractively using Automation. Unfortunately, like MAPI, the Automation interface is obscure and poorly documented. Other than the documentation that comes with Developer Studio, the only discussion I have seen of the Automation interface is in the book *Microsoft Visual C++ Owner's Manual*. What I present here is merely the tip of the Automation iceberg.

The project I have chosen to automatically rebuild in this chapter is the PerlX Automation DLL introduced in a later chapter.

What Is Smoke Testing?

Smoke testing will be of particular interest if you are familiar with the work of influential author Steve McConnel, whose books on software development have become essential reading for programmers. If you are not familiar with smoke testing, see p. 405ff in McConnel's *Rapid Development* for a discussion of the concept behind the

software smoke test. See also *Software Project Survival Guide*, especially p. 205ff.

In a typical project, one or more developers work on particular assigned tasks, perhaps checking code in and out of a source code control system. Developers can work on different modules in complete isolation from one another.[1] Eventually, all the separate code from different modules is built into the entire program. Hopefully, when this build occurs, everything works fine, but if problems appear, it's debugging time. Teams want to do this monolithic rebuild often (McConnel recommends daily smoke builds), to know the state of the code as a whole. A build can then be handed off to a separate testing team.

A drawback to the monolithic rebuild is that it's tedious. Instead of one DLL or one executable, the entire project must be rebuilt, which often requires the assembly of tens or hundreds of modules. Wouldn't it be nice to go home and have it done by the time you get back the next day? In this chapter I show you how to use Automation to run DevStudio noninteractively to rebuild the project.

First, a quick review of how Visual C++ organizes files: In Visual C++, each module, such as an EXE, a DLL, or an ActiveX control, is organized into projects. Each project contains the necessary files to build the module. Projects depend on one another. An executable can depend on a DLL, for example, which means that DevStudio will not rebuild the executable until the DLL is up to date. If you set up the project with the right dependencies, the root project of your hierarchy can be rebuilt and cause a cascade effect that rebuilds every piece of the overall program. All you need to do is touch off a rebuild of the master project.

But what about modules that don't belong in DevStudio? What if you have a heterogeneous mixture of development tools? If you are like me, you use Delphi (or C++ Builder) to build your GUI front ends and Visual C++ to do the meaty backend work. How can DevStudio rebuild a Delphi program? Or a MicroFocus COBOL program? Almost any development platform supports some type of command-line rebuild (even if it is only a Perl rebuild program as developed in the next chapter) and can be added to DevStudio using the custom build feature. This support for a command-line rebuild allows you to have the best of both worlds: interactive development in the tool itself and automated rebuilds from DevStudio. Most development environments don't work well with code from other development tools, but Visual C++ can be extended to recompile code from other languages and tools.

Here is an example of how you would set DevStudio up to automatically rebuild Delphi programs noninteractively:

1. Create a batch file that will automatically rebuild the project by uttering the magic command line incantations. All you need to do with Delphi is have it recompile the DPR file with the command `dcc32 myproject.dpr`.
2. You can also give command line parameters to control optimization (see the Delphi manuals for full details.) On my Win95 system, I have to manually set the PATH to

Delphi's `bin` directory, too, because the path to the `bin` directory is over 20 characters long and I can't fit it in the `AUTOEXEC.BAT PATH` statement.

3. Create a new Win32 application or DLL project and insert it into your workspace.
4. Add the `DPR` file from the Delphi project. Visual C++ will let you do this even though it does not know what a DPR file is.
5. In the Workspace window, switch to File view and find the `DPR` file. Right click and pick Settings.... Visual C++ will automatically switch to the Custom Build, since it has no idea what a `DPR` file is.
6. In the Description edit box, type `Recompile Delphi Program`. This description appears in the Output window when you rebuild.
7. In the Build command(s) edit control, type the name of the batch file you created in Step 1. (This list box is weird. To edit an existing entry, press F2 just as you would to edit an Excel spreadsheet cell.)

Developer Studio Automation

Developer Studio has a little known Automation interface called `MSDEV.Application`. It is poorly and incompletely documented. Using it is a lot like throwing darts blindfolded: you try something, then peek out from under the blindfold to see how it worked.

What is known about DevStudio's Automation interface can be found online. In the 97 version, using InfoViewer, you can find the information on the object model in the InfoView window by drilling down to Developer Products > Visual C++ > Developer Studio Env. User's Guide > Automating Tasks in DevStudio > Developer Studio Objects. If you have the 98 edition, you can drill down to it through Visual C++ Documentation > Using Visual C++ > Visual C++ User's Guide > Automating Tasks In Visual C++.

Code to Automatically Rebuild a Project

The Developer Studio rebuild program is shown in Listing 5.1 Some notes on the code follow the listing.

Listing 5.1 Developer Studio Rebuild Program

```
# Perl code to rebuild a project in Developer Studio
#
# The variable $dswfile needs to be set to the full path
```

```
to the
# project you want to rebuild. This script is a bare
bones
# skeleton that shows you the basic steps required to
rebuild
# a project using Perl and Automation. It could be made
much fancier
# by, for example, e-mailing the results to someone.


use Win32::OLE;


$dswfile =
"d:\\scott\\work5\\autobook\\cdrom\\PerlX\\PerlX.dsp";


print "\nRebuilding project $dswfile...\n";


# In addition to just building the project, we also keep
track
# of how long it takes.


$begin = time;


($sec, $min, $hour, $mday, $mon, $year, $wday, $yday,
$isdst) =
  localtime(time);
$mon++;
$year += 1900;
$date = sprintf("%0.2s/%0.2s/%0.4s", $mon, $mday, $year);
$time= sprintf("%02.2s:%02.2s:%02.2s", $hour, $min,
$sec);


print("Begin  at $time on $date\n");


# Developer Studio's Automation interface is
MSDEV.Application.
```

```
$ds = new Win32::OLE 'MSDEV.Application'
  or die "Couldn't create new instance of DS!";


#Uncomment this to make DS visibile as it does the
following steps
#$ds->{Visible} = 1;


$p = $ds->Documents->Open($dswfile);


$ds->RebuildAll();


$ds->Quit();


$end = time;


($sec, $min, $hour, $mday, $mon, $year, $wday, $yday,
$isdst) =
  localtime(time);
$mon++;
$year += 1900;
$date = sprintf("%0.2s/%0.2s/%0.4s", $mon, $mday, $year);
$time= sprintf("%02.2s:%02.2s:%02.2s", $hour, $min,
$sec);


print("Ending at $time on $date\n");


$e = $end - $begin;


print("Elapsed time = $e seconds\n");
```

Code Notes

`MSDEV.Application` is the Automation interface for Developer Studio. You can find out more about it using the online help.

The line:

```
$ds->Documents->Open($dswfile);
```

opens the project. Note that this line must refer to the `DSP` file, not the `DSW` file. If you try to open a `DSW` file, you will get an error message. In order to automatically recompile projects using my method, you must establish dependencies among projects. In Developer Studio, establish dependencies for the active project by choosing Project/Dependencies… and checking which projects the active project depends on. You can nest these dependencies: if one project depends on another project that has dependencies, DevStudio will rebuild all the projects inside-out.

The line:

```
$ds->RebuildAll();
```

simply rebuilds the project. I chose to totally rebuild every file to bring it up to date. I envision this automated procedure running when there is no pressure to finish, so there is no reason not to clean out all the intermediate files and rebuild the entire project. (I'm a little biased because I've been the victim of incomplete rebuild bugs in the past, which has made me somewhat paranoid. I'm prone to do "unnecessary" rebuilds just to make sure there is no un-recompiled code in the project.)

Concerning the line:

```
$dswfile =
"d:\\scott\\work5\\autobook\\cdrom\\PerlX\\PerlX.dsp";
```

I intentionally use the cumbersome double quote and double backslash as a way to make it self-evident that something really bizarre is going on. If I am using a filename internally in Perl, I will always use the forward slash. But if I am creating a filename to pass to the shell, I will use this cumbersome and attention-getting, ugly style to make the pathname stick out like a sore thumb in the code. This way, I call attention to the fact that this path is special; the path will be passed to a shell or another program and not used internally in Perl. I don't want this fact to blend into the background.

Notes

1. Indeed, this isolation is often a preferred way to work. For testing a particular type of DLL in a project, I wrote a console program to load and call its functions so I would not have to load the entire server (of which the DLL would eventually be a part) just to run the DLL. It was much quicker and easier to run the

test program during development.

Chapter 6

Automatically Rebuild Anything

The ability to quickly rebuild applications is a task at which Perl excels, as this chapter proves. Upon looking at all the different file formats used to rebuild software development projects, I conclude that using Perl to rebuild my assorted projects is an effective way to use Perl's scripting capabilities.

File Formats Everywhere!

In the previous chapter, the topic was rebuilding a Developer Studio project. In this chapter, I discuss the more general problem of rebuilding projects in any format. The problem today is that so many different development environments and tools exist that the number of ways to rebuild projects is staggering. Also, the industry has not agreed on a standard. Some tools have no project rebuilding features at all, like the Java JDK. (The JDK doesn't come with any project management tools like `make`, but the compiler does have a primitive ability to recompile dependencies.) Some Windows tools use build utilities loosely based on UNIX's `make`. Borland has a fairly close but extremely buggy clone of `make`, and Microsoft has a vaguely make-like thing called `nmake`, which uses a syntax different from that of traditional make. Some tools use binary file formats that can't be manipulated outside of the tool. AInd tool vendors have been known to change their method for rebuilding projects between releases! The situation resembles the aftermath of the tower of Babel, with many different and incompatible ways of doing the same thing.

You could use `make`. But which one? Microsoft's `nmake` is a different and incompatible dialect from other `makes`. Borland's `make` is buggy, so much so that I have trouble using it on even simple projects. (Its familiar "command too long" error is infuriating, especially since this message apparently has nothing to do with the length of the command line.) UNIX `make` clones are plentiful, with GNU, public domain, and commercial implementations. Which `make` clone should you use? Will it be available on all platforms and work exactly the same way on each?[1]

As an alternative, Perl can present you with a standard way of rebuilding projects. These rebuild scripts can be made platform independent with a little extra work.

One problem with using batch files under Windows 9x to automate rebuilds is the path length limitation. On NT, you do not have this problem, since the path is virtually unlimited in length. Windows 9x imposes severe restrictions on the path. Each development environment you install wants to put its `BIN` directory on your path (at the very least, and most want to put more than that). It's mathematically impossible (especially with long filenames) to cram all the directories you need onto the path to run the various development tools you'd typically need.[2] But, on the other hand, if you put a `SET` command at the beginning of your rebuild batch file (in order to place the `BIN` directory of the specific tool you need on your path), you run into the problem that every time you run the batch file, you prepend the new path and eventually overflow the path. The Perl rebuild solution doesn't have this problem because any changes made to its environment will be discarded when the script exits.

Using Perl to Automate Rebuilds

Luckily, it is easy to create a Perl program that will rebuild projects for you. In the last chapter, I talked about Developer Studio specifically, but the idea in this chapter is to create a command-line rebuild script that works with any command line tool in Perl. For illustration purposes, I will be using a Perl script that rebuilds a Java program. You can use the same concepts to create a rebuild script for any command-line development tool.

Features that make the Perl rebuild interesting include:

- Perl can perform strange tasks IDEs (Integrated Development Environments) can't do: A case in point is my Java example below. I search the Java files for all the public symbols.[3] Try teaching your IDE to do that! Perl gives you an unparalleled flexibility to do things that IDE inventors would never have even thought of.
- Perl can run noninteractively. Unlike the Developer Studio rebuild developed in the last chapter, this rebuild can noninteractively rebuild *any* project.
- Perl can send you e-mail: using the `smtpmail` function developed in an earlier chapter, the rebuild program can send you e-mail when it is finished. You could even modify this program to send you the actual compiler output using techniques developed for the backup program in Chapter 3. (I don't know of an IDE that could do this. Visual C++ could run a program that sends e-mail as a post-build step, but I don't know how it could get information about the

compile itself.)

- Perl can time how long the rebuild takes.

The following example shows how to automatically rebuild a Java project. Java is particularly annoying in Windows 9x, where the path length is limited, because every Java SDK and add-on (such as a vendor-supplied JDBC class) wants to add a long pathname to your path. As mentioned earlier, the Perl program gets around this problem, so you do not need the JDK `bin` directory on your path.

Code

The Perl rebuild script is shown in Listing 6.1. Some notes on the code follow the listing. The Java sample program `hello.java` referenced in the rebuild script is the following Hello, World program:

```
import java.io.*;

class hello {

  public static void main(String[] args) throws
IOException {

    System.out.println
("Hello, Java program compiled by a Perl script!");

  }

}
```

Listing 6.1 Rebuild Script

```
#############################################################
############
# Perl Rebuild Script
#############################################################
```

```
############


# set the CLASSPATH for Java -- note on Windows, we have
to use
# \\ instead of / because this is going out into the
shell
# and is not part of Perl which doesn't care


# Obviously, change this to where the JDK is installed
$ENV{"CLASSPATH"} .= ";d:\\jdk1.1.6\\lib\\classes.zip";


############################################################
############
# compiler options
# (note: create as many of these as needed for whichever
# platforms, etc this has to be compiled on, and comment
# out all but the one in use at the moment)
############################################################
############


$javaroot = "d:\\jdk1.1.6\\bin";
$javacompiler = "javac.exe";
$javaflags = "-deprecation"; # full details on Bad Things


############################################################
############
# project source files
# (note: these are compiled in order listed)
############################################################
############


@sourcefiles = ("hello.java");


# rebuild all files
```

```
($sec, $min, $hour, $mday, $mon, $year, $wday, $yday,
$isdst) =
  localtime(time);
$mon++;
$date = sprintf("%0.2s/%0.2s/%0.4s", $mon, $mday,
$year+1900);
$time= sprintf("%02.2s:%02.2s:%02.2s", $hour, $min,
$sec);


print "Build started at [$date $time]\n";


for (@sourcefiles) {
  $rc = system("$javaroot/$javacompiler $javaflags $_");
  if ($rc != 0) {
    die "ERROR: rc=$rc from compile of $_\n";
  }


  # You can also do other processing such as this...
  open(SRC, "$_") or die "Can't open file $_ for
reading";
  @output = grep(/public /, <SRC>);
  print "[$_ public symbols]\n";
  print @output;


}


($sec, $min, $hour, $mday, $mon, $year, $wday, $yday,
$isdst) =
  localtime(time);
$mon++;
$date = sprintf("%0.2s/%0.2s/%0.4s", $mon, $mday,
$year+1900);
$time= sprintf("%02.2s:%02.2s:%02.2s", $hour, $min,
$sec);
```

```
print "Build finished at [$date $time]\n";
```

Code Notes

The code is very simple and straightforward. For each source file in the list `@sourcefiles`, the `$javacompiler` is invoked. With Java, of course, there is no link edit step. Once you compile a Java file, that's all you have to do with it. A language like C would require a final link edit step to create an executable (or some other module like a DLL) out of the source files.

Ways to improve this script include:

- You could create logic to look at the filename's extension and invoke a different compiler based on what the extension is. The way I have the program set up now, you must change it for each language.
- You could add a `make`-like feature that recompiles the file only if its timestamp is newer than the result of the compilation.

Portability of this script can be achieved by changing the `$javaroot`, `$javacompiler`, and other variables. Comment out one platform's set of variables and uncomment another platform's variables. On a UNIX system, you could change the paths to be relative to the root directory (`/`) instead of a drive letter, and you could change the path delimiter.

The path delimiter must be `\`, not `/`, because it is being passed to the shell via `system()`. Perl doesn't care, but the shell does. Since Perl treats `\` as a special escape character in strings (think `\n`), it must be escaped and written as `\\`.

Notes

1. There is also now a "make" Perl module that emulates UNIX make. It would be a good thing to standardize on if you wanted to use UNIX make on all platforms.
2. I don't know how typical I am, but I looked on my systems and saw I had Developer Studio, Delphi, Borland C++, the JDK, and MicroFocus COBOL, which I use regularly, and five or so other tools I use rarely. Most of these either want to pollute my PATH or are annoying to use without being on my PATH. Note that I am not counting other tools like DB2 that are not strictly development tools but that also need to pollute the PATH.
3. I can't even remember why I wanted to do this.

Chapter 7

Creating C++ Add-Ins for Perl

It is said that Perl makes easy things easy and hard things possible. For impossible tasks, sometimes you must call on something more powerful, such as C++, which can do things no other language can do. In this chapter, I'll explain why extending Perl with Automation servers (written in Visual C++) is a good alternative, and I will develop an Automation server that gives Perl some added capabilities, including the ability to use a GUI interface.[1]

Extending Perl with Automation Servers

As I've discussed in previous chapters, Automation is a paradigm in which functionality is separated from the use of the functionality. A server provides certain well-defined functionality to any client that wants to use it, and a client uses whatever functionality it needs from different servers to create a solution to a problem. Functionality exists in packets called objects. Each object has an interface that defines both methods and properties. An object has enough methods and properties to do whatever it is designed to do. Some objects are big; some are tiny.

Any given object can be both a client and a server. Many programmers (particularly those who come from other areas of computing that have no real notion of objects) have trouble understanding that the role of an object (or anything else in a client/server system, even a computer itself) changes based on what the object is doing. An object can be a server to one client and a client to another server—even at the same time! What is a *client* and what is a *server* depends on context.

Automation servers are not the *only* way to extend Perl. You have several choices for creating a functionality not present in the core of Perl. You can write the new functionality in the Perl language itself as a new module. You can also use the XS interface to extend Perl using C modules. Although XS was once a black art reserved only for the Perl elite gurus, documentation on how to use XS is becoming more widespread, and XS is no longer an unapproachable alternative.[2]

Among the alternatives, however, I think a compelling case can be made for writing an Automation server in C++ and then letting Perl programs call the Automation server using `Win32::OLE`. The reasons for using Automation to extend Perl include:

- If you know C++ and Windows software development, you probably already know how to create an Automation server in Visual C++.
- If you *do not* know C++ and Windows software development, building an Automation server is perhaps the easiest technique to learn. The alternative, XS, is much harder to learn from scratch with no previous experience.
- Using Visual C++'s ClassWizard, maximum code can be built with minimum effort. I assume the readers of this book do not have ample spare time.
- Creating an Automation server allows you to reuse existing C++ code such as dialogs. Once you see just how reusable your code can be, you may make all of your code into Automation servers and arrange it using an Automation client language.
- Automation servers can be reused with any language that can be an Automation client (Visual Basic, Delphi, and other languages).

Automation servers can be written in almost any language, as long as it speaks COM. The example in this chapter uses Visual C++, but similar servers could be built using Delphi and many other tools. Space and time do not permit creating a Delphi example extension.

In this chapter, I will create an Automation server called `perplex` and add an interface called `DoStuff`. This interface will demonstrate a property, a method, and a method that displays a dialog box. Once this Automation server is developed, I'll show you how to create a sample Perl program that will use the server. I will also demonstrate how to build Automation clients in other languages.

Testing Automation Servers with Perl

The sample program in this chapter is an example of how you can use Perl to test Automation servers. Perl is the perfect language for testing Automation servers, particularly Automation servers that are not GUI driven and tend to be used noninteractively. Perl allows you to write good test programs to exercise a component fully.

For a noninteractive component, such as something that mucks around in the low-level bowels of your system, the typical Automation testing strategy doesn't adapt very well. Before Perl, a programmer would typically write an Automation server in a friendly Automation client language like Visual Basic that allows quick construction of an Automation testbed. The application would be little more than a form with a "go" button and some fields for parameters. This solution was fast, and it offered the friendliness of a language such as Visual Basic, but it lacked the ability to test automatically.

Perl scripts can be executed automatically without human intervention, and they

can therefore be part of a larger suite of tests. In a later chapter, you will see that it is easy to integrate a Perl script with a Developer Studio project.

Creating the `PerlX` Automation Server

The basic steps in creating an Automation server are conceptually easy to understand. The mechanics of creating the server are specific to the development tool you use. In this chapter, I show you how to create an Automation server DLL using Visual C++. The same basic conepts will apply to other tools, but the mechanics will be totally different. If you are not using Visual C++, consult your tool's reference on how to create Automation servers.

This chapter creates what is known as an "in-process" server, a DLL that will be loaded into the same address space as the running program. The distinction between these in-process servers and other types of Automation servers are beyond the scope of this book. Consult a good COM reference for details on other types of Automation servers. For our purposes, a DLL-based server is the easiest type to create and to explain.

Each Automation server has a name, such as `PerlX` in this example. Each Automation server provides one or more interfaces, which also have names, such as (in this example) `DoStuff`. To refer to a specific interface of a specific Automation server, a dot-separated fully qualified name is used: `PerlX.DoStuff`. This example provides only one interface, but an Automation server may have more than one.

Each interface in an Automation server has either properties, methods, or both, which it exports for the use of the client that is calling it. Properties generally are variables that control the operation of the server, and methods are essentially function calls that the client makes to the server and from which the server can return the results of whatever it does. The following example has one property and two methods.

Constructing an Automation server is a matter of creating an Automation DLL and naming it, and then adding to it interfaces with their properties and methods. You can easily do this in Visual C++. In fact, "Visual" C++ is much better suited for (noninteractive) server programming than it is for GUI programming! The "visual" in the name is a misnomer. In my career, I have primarily used Visual C++ for the development of servers such as Automation DLLs and TCP/IP servers; I have found it to be an excellent environment for developing this sort of software, and I enjoy using it.

In addition to creating new Automation server DLLs, it is also easy to create Automation interfaces for existing function libraries by creating an Automation DLL that provides methods corresponding to API functions in the libraries. A good way to make a code library portable between UNIX and Win32, for example, is to create the core library in Standard C or C++ and create interface wrappers on either platform to control how it is called. On Windows, this wrapper could be Automation. To port the code, all you need to do is rewrite the interface, since the standard code should be portable as

is.

To begin constructing the example PerlX Automation server: Open Developer Studio (or close any open workspaces), and select File/New… (See Figure 7.1.)

Figure 7.1 Creating a Project in Developer Studio

On the next panel (see Figure 7.2), make sure to check Automation. The other choices are up to you, but the defaults are typical.

Figure 7.2 The MFC AppWizard

Adding a `DoStuff` Interface to PerlX

Once you have generated a new DLL project, all you have is the boilerplate code. The DLL doesn't actually do anything yet. The Automation server PerlX exists, but it doesn't expose any interfaces that clients can use.

The next step is to add a new interface for the Automation server to expose the interface called `DoStuff`. Select Insert/New Class… and fill in the dialog as shown in Figure 7.3.

Figure 7.3 Creating a New Class

The `DoStuff` class inherits the MFC class `CCmdTarget`, which is the class for mapping messages in MFC between events that happen outside of your classes to the methods in your classes that handle them. In this case, the `DoStuff` interface needs to map incoming COM events to `DoStuff`'s methods.

Most notable here is that the class has been derived from `CCmdTarget`. `CCmdTarget` is an interesting class. In the original MFC, it did not exist. The story of `CCmdTarget` is the story of Windows' awakening to the possibility of noninteractively running programs. The original MFC introduced the idea of a *message map*, a mechanism by which messages are received by an application and dispatched to a class method that will handle them. Originally, the only messages an application could receive (in Windows 3.x) were window messages, so the message map mechanism was built into the `CWnd` class. The idea of abstracting message mapping into an ancestor class didn't exist yet because there were no other types of message maps besides those handling Windows messages. The idiom of a message map proved to

be elegant and extremely useful, so much so that the functionality that supported it "bubbled up" a layer in the MFC class hierarchy to become `CCmdTarget`. Now, message maps could be used for more than just window messages. Automation, for example, uses dispatch maps that route `IDispatch` interface calls to the methods that support them. The general idiom of message maps has become a powerful part of Win32 programming over the years.

Also note that Creatable By Type ID has been checked. The Creatable By Type ID option makes it easy to create objects from Perl. The type ID is a string such as `PerlX.DoStuff`, which the COM subsystem maps to a CLSID. A CLSID is a long number, the class ID, which uniquely identifies your Automation interface. The mechanism for how this is done is beyond the scope of this book, but it involves the system registry. The relationship between the Type ID and the CLSID is similar to the relationship between Internet hostnames (www.somehost.com) and IP addresses (127.0.0.1). The point is to replace long strings of numbers with easier-to-use words.

When you're through with the New Class dialog, you should have a `ClassView` window that looks something like the window shown in Figure 7.4.

Figure 7.4 The ClassView Window

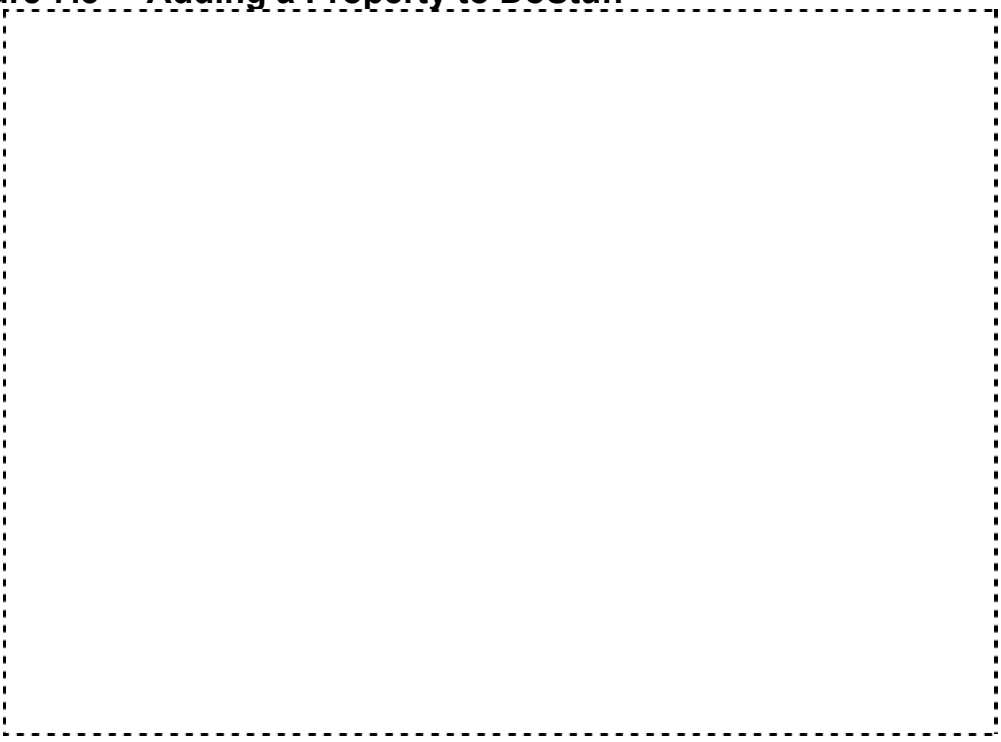
This Automation DLL now has an interface, `IDoStuff`, but it exposes no properties or methods yet. It's important to note the distinction between `IDoStuff` and `DoStuff`. `IDoStuff` is the interface class the Automation client sees, and `DoStuff` is your internal implementation of the functionality of that interface. MFC and the ClassWizard do the hard work of mapping the interface to your implementation for you.

Adding a Property to `DoStuff`

The first thing to add is a property. In this example, a string called Storage will be added. Properties are essentially variables that are exposed to the Automation client.

Select `DoStuff` (*not* `IDoStuff`) and then choose View/ClassWizard from the menu. Go to the Automation tab. Click Add Property… and fill out the dialog box as shown in Figure 7.5.

Figure 7.5 Adding a Property to DoStuff



Note the data type is `CString`. This feature is one of the magical wonders of the ClassWizard that makes Automation the winning choice for mixed language programming. You can manipulate an MFC CString in your code, and the Automation code automatically generated by the ClassWizard handles the details of converting the string to and from an OLE string data type. (If you click to expand `IDoStuff` in the ClassView window and then double click Storage, you'll see a line that says something about BSTR Storage. BSTR is the OLE string datatype that you never have to deal with.)

You can now use this property in Perl in this way:

```
use Win32::OLE;
$mything = new Win32::OLE 'PerlX.DoStuff' ||
  die "Couldn't create new instance of PerlX.DoStuff";
$mything->{'Storage'} = "I need to store a string";


print "In storage: " . $mything->{'Storage'}
```

Adding a Method to `DoStuff`

Properties are nice, but they don't do much. Methods, on the other hand, are subroutines that can do anything you can do on the computer from initiate an Internet request to rebooting the machine. A method is a C++ subroutine that can be called from the Automation client, the Perl program. The possibilities of what you can do in the method are unlimited.

I will add a method to DoStuff that is a wrapper to the `MessageBox` Windows API function, the staple of Windows debugging. This is a simple way to add a GUI to your Perl program that doesn't involve dialog box programming.

First, go back to the ClassWizard and click Add Method…. Then fill out the dialog as in Figure 7.6.

Figure 7.6 Adding a Method to `DoStuff`

Again, the ClassWizard handles most of the gory details for you. All you need to do is click on Edit Code after you have created the Say method, and add this line:

```
MessageBox(0, Message, "PerlX", 0);
```

Adding a Dialog to `DoStuff`

Properties are nice, and methods are better, but adding custom dialog boxes will really make your programs dazzling.

First, add a method to the class named `GetName` that takes no parameters and returns a BSTR. (You still don't have to deal directly with a BSTR, since the glue code will convert this to an MFC CString internally and convert it back at the end of the method.)

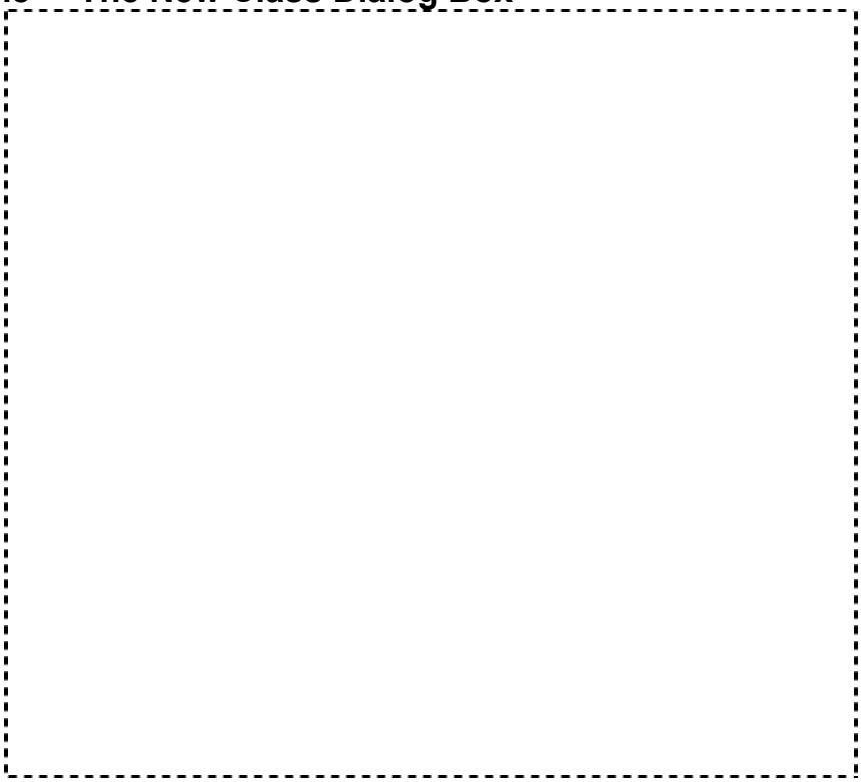
You can now create a dialog. Go to the ResourceView and select PerlX resources, then right click and select Insert…. Choose Dialog and click New. You should see a basic dialog with OK and Cancel buttons.

Try putting a static text control on the dialog, and try changing the label. Developer Studio will interrupt you with the message shown in Figure 7.7

Figure 7.7 Creating a Class

You want a new class, so click OK. Fill out the New Class dialog as shown in Figure 7.8.

Figure 7.8 The New Class Dialog Box



Once you click Ok for this dialog, the code you need will be generated automatically. Now go back to the ResourceView and edit the dialog. Make it look something like Figure 7.9.

Figure 7.9 Configuring an Edit Field

The main thing the dialog must have is the edit field. I use the default identifier `IDC_EDIT1` for it. Now go back to the ClassWizard and the Member Variables tab (see Figure 7.10). You should see `IDC_EDIT1` in the list. Double click `IDC_EDIT1` and fill out the dialog.

Figure 7.10 The MFC ClassWizard Member Variables Tab

Now you're ready to complete the `GetName` method of `IDoStuff`. `GetName` will display this dialog and return the string to the caller. Return to the ClassWizard, find `DoStuff::GetName()`, and implement it like this:

```
BSTR DoStuff::GetName()
{
            CString strResult;
            // TODO: Add your dispatch handler code here
            GetNameDlg g;

            if (g.DoModal() != IDOK) strResult =
"[error]";

            strResult = g.m_name;

            return strResult.AllocSysString();
}
```

Be sure to add:

```
#include "GetNameDlg.h"
```

to the top of `DoStuff.cpp` so it knows what a `GetNameDlg` is.

Now that you've finished, you can get someone's name in Perl by using this code fragment:

```
use Win32::OLE;

$mything = new Win32::OLE 'PerlX.DoStuff' ||
  die "Couldn't create new instance of PerlX.DoStuff";

$name = $mything->GetName();
print "Your name is $name\n";
```

This general technique can be used to create any kind of dialog your Perl program might require. Once written, the same dialog can be run from *any* Automation client, which is what makes this technique so powerful. You're creating building blocks for applications. Perl is the glue that holds the building blocks together.

Using the PerlX Property, Method, and Dialog

The following Perl code uses all three parts of the `DoStuff` interface:

```
use Win32::OLE;

$mything = new Win32::OLE 'PerlX.DoStuff' ||
  die "Couldn't create new instance of PerlX.DoStuff";

$name = $mything->GetName();

print "Your name is $name\n";

$mything->Say("Hello, World!");

$mything->{'Storage'} = "I need to store a string";

print "In storage: " . $mything->{'Storage'}
```

Before you can use this code, though, you need to register the control. The registration process makes the control available on the system by making all the magic registry entries.

In Developer Studio itself, to register the control you can select Tools/Register Control from the menu. If you want to register the control on another machine, go to Tools/Options and select the Tools tab. Register Control will be the first item (unless you have customized the tool settings), and you can see the command is `regsvr32.exe` in the system directory.

Calling `regsvr32.exe` multiple times when the DLL is already registered is not a problem. You could add a line in your program to always register the DLL. If you think that is a grossly inefficient use of computer resources, you could use `Win32::Registry` to see if your DLL is loaded. If you take that approach, you should be sure to try reloading the DLL if the new `Win32::OLE` call fails. The object creation could fail for reasons such as a corrupt registry, and reregistering would clear that up.

You can look in the registry under `HKEY_CLASSES_ROOT` for your object (the example `PerlX.DoStuff` interface would be listed under `HKEY_CLASSES_ROOT\PerlX.DoStuff`), and retrieve the default value, which is just the name of the interface again.

I will not attempt a full discussion[3] of how the entry for the interface `HKEY_CLASSES_ROOT` maps to a DLL name. The process involves looking up the CLSID (the object's class ID) in `HKEY_CLASSES_ROOT\CLSID`. The GUUID[4] generated for the CLSID is an entry under this branch of the registry. If you find your CLSID under that branch, at least for the kind of DLL we're creating in this chapter, you'll find a key called `InProcServer32`, which has as its default value the path to your DLL. This layer of indirection is necessary to support "dual interface" Automation servers: languages like C++ tend to use the CLSID directly to load an Automation DLL, where Perl and Visual Basic use the human-readable name and translate that to a CLSID.

Distributing a Visual C++ 6.0 Automation Server

If you use Visual C++ *version 6*, you should be warned that you will not be able to distribute your Automation DLL to another computer without also distributing a program to register the DLL on the new system. Why? Because Visual C++ 6.0's DLLs are not backward compatible with older versions! Surprise! They changed something in how C++ symbols are exported from the DLL, and DLLs from version 6 are incompatible with earlier programs that try to call functions in them. (The reverse is true, too: you can't call version 5 DLLs from a version 6 program without relinking. This limitation seems to be limited to C++ DLLs only and doesn't affect C DLLs that use the `stdcall` calling convention and don't use C++ name mangling. This issue only applies to C++ DLLs which use C++-style entry points, which are name mangled. Unfortunately, Automation DLLs seem to be.)

This problem shows up in the fact that the `regsvr32.exe` program, which comes with Windows 9x and NT, can't read version 6 Automation DLLs. `Regsvr32.exe` is the utility program that registers Automation servers on the system.

This issue happens to be one of the silent changes that will not show up until you actually send your Automation DLL to another machine and it fails to register. Visual C++ 6 upgrades *your* `regsvr32.exe` on *your computer* silently to a new version that works with version 6 Automation DLLs. But no shipping version of Windows 95, 98, or NT 4 (up to at least SP3) has a `regsvr32.exe` program that can register version 6 DLLs! So when you send your software elsewhere, you get a big surprise. I guess Microsoft is trying to add a little variety to your life by requiring you to debug incompatibilities.

Luckily, `regsvr32.exe` is not much of a program. All it does is call one entry point of the DLL. It is easy to write your own version of `regsvr32.exe` in Visual C++ version 6 and create a version 6 compatible `regsvr32.exe`.

Create a console-mode Win32 program in Visual C++ 6 and use this code, which I found in a Microsoft Knowledge Base article. The code here does the same thing that `reqsvr32.exe` does.

```
#include <iostream>
using namespace std;

#include <windows.h>

int main(int argc, char* argv[]) {

  if (argc != 2) {
    cout << "Usage: sbregsvr [Automation DLL]" << endl;
    exit(999);
  }

  HINSTANCE hDLL = LoadLibrary(argv[1]);

  if (NULL == hDLL) {
    // See Winerror.h for explaination of error code.
    DWORD error = GetLastError();
    cout << "LoadLibrary() Failed with: " << error <<
endl;
    exit(999);
  }

  typedef HRESULT (CALLBACK *HCRET)(void);

  HCRET lpfnDllRegisterServer;

  lpfnDllRegisterServer =
    (HCRET)GetProcAddress(hDLL, "DllRegisterServer");

  if (NULL == lpfnDllRegisterServer) {
```

```
    // See Winerror.h for explaination of error code.
    DWORD error = GetLastError();
    cout << "GetProcAddress() Failed with " << error <<
endl;
    exit(999);
  }

  if (FAILED((*lpfnDllRegisterServer)())) {
    cout << "DLLRegisterServer() Failed" << endl;
    exit(999);
  }

  return 0;
}
```

Calling `PerlX.DoStuff` from Other Languages

Once you have completed writing a piece of functionality as an Automation DLL, *any Automation client* can use it. I will present samples of using `PerlX` from Visual Basic, Delphi, and C++.

Most applications tend to have several core components with highly specialized functionality that relate to what the application *does.* The rest of the application is generally a shell that drives the application's use of that functionality. The shell can either be a GUI interface for a human user, or it can be some other kind of interface for another piece of software to call.

Automation allows you to design software in which the core functionality exists as Automation components. Then, any scripting language or GUI builder can be used to glue these components together into an application. The functionality itself can be reused in different ways, both from interactive panel-driven programs and from automated scripts. Separating what the program *does* from how you do it is the underlying key to Automation.

Calling `PerlX.DoStuff` from Visual Basic

Here is a subroutine written in VBA that works the same as the sample Perl program.

```
Public Sub TestDoStuff()
```

```
    Dim x As PerlX.DoStuff

    Set x = CreateObject("PerlX.DoStuff")

    MyName = x.GetName()

    x.Say ("Hello, " & MyName & ", From VB!")
    x.Storage = MyName
    x.Say ("Stored: " & x.Storage)

End Sub
```

Before I wrote this code in Microsoft Word 97, I opened the Visual Basic for Applications development environment and selected Tools/References, then clicked Browse…, and then added the TLB file for PerlX. (Developer Studio automatically generates a TLB file for you.)

The TLB file is a type library that tells an Automation client all about what to expect from an interface like `PerlX.DoStuff`. You do not strictly have to use the TLB file, but using it makes developing in the Visual Basic code editor a much more enjoyable experience because VB knows all about your object and can show you the properties and methods when you start typing the object's name. Running your VB subroutine is much faster since the runtime environment has advance information on the objects you are using and can plan ahead. VB does not have to look up all properties and methods at runtime.

Calling `PerlX.DoStuff` from LotusScript

The Office 97 bias in this book is largely one of expediency. I have a shelf with about 10 VBA Office 97 programming books and no Lotus SmartSuite programming books. LotusScript is a powerful language that seems to be similar to Visual Basic for Applications in most respects. Since LotusScript is part of Lotus' e-mail and groupware package, it can be extended from the desktop to the entire enterprise. Unfortunately, you'll find few educational resources that will teach you how to program in LotusScript. What I present in this section came strictly from trial and error.

The LotusScript code that calls the PerlX object is very similar to the Visual Basic code discussed in the preceding section, but you'll find a few differences, most notably the lack of direct support for creating a `DoStuff` object. Instead, a Variant must be created and assigned the result of `CreateObject`. This code was created using Word Pro 97. I am not sure if LotusScript's support for Automation has changed since that version or not, since I do not have a later version to experiment with.

```
Sub Main
           Dim x As Variant
           Set x = CreateObject("PerlX.DoStuff")
           Messagebox "Calling PerlX..."

           myname = x.GetName()
           x.Say(myname & " is my name...")
           x.Storage = myname
           x.Say(" ... " & x.Storage & " is in storage!")

End Sub
```

Calling `PerlX.DoStuff` from Delphi

Delphi's support for Automation clients is midway between Visual Basic's and Visual C++'s. It is not quite as seamless as Visual Basic, but it is not as low-level as Visual C++. In this example, I am using Delphi 3. I am not sure if the Automation support was present in earlier versions, but it should be available in all later versions.

First, create a new project in Delphi. Then, select Project/Import Type Library, and Add… the PerlX TBL file. Delphi creates a unit that has all of the information about the interfaces the TLB describes. This Pascal unit is similar to the file the Visual C++ `#import` statement automatically generates. The compiler, either Delphi or C++, reads the TLB file and creates abstract base classes in the native language that describe the interfaces exported by the object. Creating the abstract base classes in the native language makes it easier to use the interfaces because your compiler knows ahead of time what to expect. Also you can use classes defined in a high-level language. Creating the abstract base classes in the native language also makes the process faster (if the object supports dual interfaces, which are beyond the scope of this book).

Create a main form with an edit box similar to the box shown in Figure 7.11.

Figure 7.11 Delphi Calls PerlX Form

And use this code as the button's Click method:

```
procedure TForm1.Button1Click(Sender: TObject);
var
   mything: IDoStuff;
   myname: string;
   storestring: string;
begin
     mything := CoDoStuff.Create;
     myname := mything.GetName;
     mything.Say('My name is: ' + myname);
     mything.Storage := 'Delphi stored here';
     StorageOutput.Text := mything.Storage;
     NameOutput.Text := myname;
end;
```

Of course, you must use the unit generated from the TLB file in order for Delphi to know what an `IDoStuff` and a `CoDoStuff` are. Add this line after the implementation line in your form's module.

```
uses PerlX_TLB;
```

A few words on Delphi's object model: Delphi is philosophically almost identical to Java in its object model, although Delphi approaches the model from a Pascal syntax

and Java from a C-like syntax. If you disregard the syntax, though, the conceptual underpinnings of the object model are the same. Java, syntax aside, is much more similar to Delphi than it is to C++.

You're not seeing a typo in the preceding code; that's really an `IDoStuff` variable and a `CoDoStuff` constructor. The relationship between an `IDoStuff` variable and a `CoDoStuff` constructor is the same as that between a pointer to an abstract class and a concrete instantiation of the class, or an interface (I) and an implementation (Co). Delphi, like Java, does not differentiate between a *pointer to* an object and an *instance* of an object. Since all objects are created on the free store (in other words, the heap), there's no reason to have the distinction of the difference between a pointer and an instance, since an object can never be created anywhere else *but* dynamically on the free store. Only languages that allow objects to be created somewhere besides the free store need symbolic notation to differentiate between objects in the free store (in C++, the arrow operator) and elsewhere (in C++, the dot operator).

Like Java, Delphi has the notion that when you declare a variable of a class, it is an uninitialized reference that can refer to an object of that class (in C++, you'd say a variable "can point to" an object of that class, but Delphi and Java don't have the concept of pointers). A *reference* is also called a *handle* in some circles.[5] Whatever you call it, you have to associate your uninitialized variable with an instance of a class before you can use it. That is what the constructor does in Delphi and Java. The constructor constructs a new instance of the object in the free store and binds a handle to it.

The key difference between a C++ constructor and a Delphi or Java constructor is that in C++ the constructor has nothing to do with allocating memory for the object itself (although it can allocate memory for the object's internal use). In Delphi and Java, the constructor allocates memory for the object itself as well as performing the object's internal initialization. To C++ programmers, Delphi and Java's constructors are strange and take some getting used to. To Java and Delphi programmers, writing C++ code is like trying to run underwater, because it is easy to lose sight of what you're trying to do for all the low-level details you have to keep up with.

One significant difference in philosophy between Delphi and Java is that Java is garbage collected. That means you never have to bother with allocating or freeing objects. The language itself does *both* for you automatically in the background. In Delphi, you are responsible for destroying any objects you create when you are through with them. Delphi has no automatic garbage collection. In Delphi, you don't ever keep track of where the object is created, but you must keep track of the object's lifetime yourself.

Overall, the philosophy of having all objects dynamically allocated in an object-oriented programming language and the elimination of explicit object pointers greatly simplifies many aspects of object-oriented programming. This is the main reason why

Java is perceived as simpler than C++: just eliminating the possibility of static objects or those on the stack greatly simplifies the entire language.

`IDoStuff` is a pointer to a `DoStuff` interface, and `CoDoStuff` creates a concrete implementation of `DoStuff` for it to point to.

Finally, I want to point out there is no particular reason to create a GUI program in Delphi. I just created a GUI program as an example. If you use the right library units, you can easily add Automation client support to a command line program written in Delphi.

Calling `PerlX.DoStuff` from C++

As a final `PerlX` example, I present a C++ client. Actually, "C++" is a bit of a misnomer for this client. The compiler that we and Microsoft call "Visual C++" has mutated into "Visual COM." Visual C++ has a basis in C++, but Visual C++ really supports C++ *and* one other thing: binary reusability. I've long considered the lack of a binary standard C++'s biggest weakness, and one that's almost insurmountable. Standard C++ is only source code compatible. Object code from two compilers can't be linked together in a standard way.

With C, you didn't notice the lack of a binary standard as the problem, first because C's linkage is so simplistic (in comparison to C++; and remember C only guarantees 6 bytes of unique external identifiers[6]), and second because everything on the system used C linkage anyway.[7] With C++, the lack of a binary standard becomes problematic for reasons that are beyond the scope of this book. To address this problem, Microsoft has created COM and dictated that all Windows programming be done COM-style.

The Visual C++ compiler has extensions which make COM programming easier. As I mentioned earlier, COM's #1 problem in its early years was its overwhelming complexity and the primitive quality of the COM development tools.

To use the `PerlX` server, put the following code in the `main()` function of a Win32 console mode application that is set up to link with the standard Win32 API libraries:

```
CoInitialize(NULL);

try {
  IDoStuffPtr mything(__uuidof(DoStuff));
            CString StoreMe("C++ Stored this...");
            _bstr_t MyName;

            if
```

```
(mything.CreateInstance(LPCSTR("PerlX.DoStuff")) != S_OK)
{
                cout << "Can't create a DoStuff" <<
endl;
                exit(999);
             }


           mything->Storage = _bstr_t(StoreMe);
           mything->Say("Hello, World!");
           MyName = mything->GetName();
           _bstr_t out = "Storage:" + mything->Storage +
", Name" + MyName;
           mything->Say(out);


}
catch (_com_error & e) {
           cout << "COM error: " << e.ErrorMessage() <<
endl;
}

CoUninitialize();


return 0;
```

For Visual C++ to know what an `IDoStuff` pointer is, you must import the type library. Put the following code in a header file someplace:

```
#include <afxdisp.h>


#import "D:\Scott\work5\autobook\PerlX\Debug\PerlX.tlb"
rename_namespace("PerlX")
using namespace PerlX;
```

You will have to change the path to the PerlX.tlb to point to where it is on your system. This `#import` statement causes Visual C++ to emit volumes of code for you.

A BSTR, which has recently been renamed as the compiler-extension data type

`__bstr_t`, is the language-independent string data type. C strings (which Microsoft calls `LPSTR` or `LPCSTR`, depending upon what you read) are inadequate since many languages don't support variable-length, zero-terminated strings that are referred to by pointer. Visual Basic doesn't, Delphi wouldn't if Borland had not grafted support onto Delphi just to accommodate a C-centric universe, and other languages like COBOL can't. BSTR is the glue that holds Automation together because there is no standard, cross-language string data type. Usually, you will never have to deal explicitly with a BSTR in Visual C++, since it gives you the glue code automatically to convert to and from traditional `LPSTR`s and MFC's `CString` class. Occasionally, such as in this C++ example, you must directly use a BSTR, but it is normally well hidden from you by MFC and the ClassWizard.

Notes

1. As this book was being written, support for the Tk interface to Perl became available for the Win32 port. Tk is a GUI widget toolkit that comes from the UNIX world, where it was the Delphi-like rapid application development tool for the X Window system. If you think it is hard to program graphics in Windows, X is worse than you could ever imagine. Before Tk, no real alternative to low-level C code existed for GUI development. Tk is of interest both to transplanted UNIX developers, who already know it, and to people who want to create cross-platform GUIs. The material on developing GUIs using Automation servers is still of interest to Windows programmers, however, because Tk is likely going to have a learning curve, and Visual C++ is likely already to be well known.
2. Although I do not discuss XS in this book, and I say XS is harder to learn from scratch, don't get the idea that I am knocking it. I am constrained by time and space to talk about things that have not been talked about elsewhere. The XS interface is finally becoming fairly well documented, and if you need to learn it, you can pick it up. XS is a viable alternative and may be the best alternative in many instances, especially if you have C code you want to compile and embed in Perl on many different platforms. XS is actually an inteface (somewhat similar to IDL) for defining the "glue" between a Perl module and an extension library. You need to learn this extension language and write all the glue code yourself. With Automation, Perl's `Win32::OLE` module has prewritten glue for the Perl side, and Visual C++ automates generation of the glue on the C++ side. For the programmer who wants to knock out code fast, Automation requires less work.
3. If you would like more information, be sure to get a good book (see Bibliography) that discusses how COM specifically works with the registry, not just how to tweak user interface settings in the registry.

4. If you don't know what a GUUID is, don't worry about it for the purposes of this chapter. It's not important. See an advanced book such as Box's *Essential COM* for details. This discussion is for the enlightenment of COM programmers getting into Perl.
5. One of the remaining problems with object-oriented programming in general is that different people still call the same idea by different names, which makes writing a book that uses mixed languages difficult.
6. I'm reminded of a mainframe programmer who was told the 6 byte restriction on symbols in the assembler had been relaxed in a new version. He scoffed at the supposed improvement, noting "I have enough trouble thinking up 6 bytes as it is!"
7. By this qualification I mean that UNIX, and all OSes after it (Windows, OS/2, etc.), have used C as their systems programming language. All interfaces are defined in terms of C, and all other languages tend to go along with it because they have to. Now, with C++, this is not the case. All systems programming interfaces are either still defined in C, or, for Windows at least, are being redefined in terms of COM.

Chapter 8

Using Perl As a CGI Scripting Language

For the programmer, writing a CGI program is a difficult challenge because CGI programs require a kind of black-box engineering effort. The internal workings of CGI are dark and hidden, and, for the programmer, the only debugging alternative often is to try something and observe not the results themselves but whatever side effects the results cause with the web server. This chapter won't turn you into a CGI guru (entire volumes have been written to do that), but it will highlight some of the interesting things you can do with Windows, Perl, and the web.

Are Perl and CGI Synonyms?

Before I begin talking about CGI programming, I want to discuss the synergy between Perl and CGI. To someone with no UNIX background, in fact, Perl and CGI are often seen as synonyms. Many programmers don't know where one begins and the other ends. The only time they've ever heard of Perl is likely in connection with CGI programming. Perl's rise from an obscure language primarily used for UNIX systems programming to one of the most exciting and hot languages of the late 1990s came along with the web explosion in 1994-96. Why are the two languages linked?

CGI is, itself, the *Common Gateway Interface*, one of those acronyms that says nothing much about what it does. Common as opposed to what, the Royal Gateway Interface? Gateway to what? The only word in the name that says anything is *interface*. CGI is a standard interface for web servers to call external programs to manipulate input and produce output, rather than just reading and displaying static web pages.

Since CGI originated with UNIX, it is built on I/O redirection, the foundation of the UNIX philosophy. The web server reads input, starts the CGI program, and pipes the data it reads (along with some important environment variables) to the CGI program's standard input. The CGI program uses the input to create output, which it writes to its standard output. The web server then sends your output back to the browser. The real power is in what your program can do between reading input and producing output, such as querying databases and sending e-mail. CGI opens up an unlimited number of

possibilities.

A CGI program can be written in any language that can read from standard input and write to standard output. (Even if a language can't, various hacks have been invented to allow the use of the language: the most famous being "Win CGI," which actually uses INI files to pass data to and from a Visual Basic program!) There is no practical restriction on the language that can be used.

Perl happens to be one of the most widely used CGI programming languages. Despite Perl's intrinsic merits for CGI processing, I assume Perl was adopted by early web server administrators (who were also the programmers who wrote CGI applications, since specialization had not developed yet) because it was a rapid application development language with which the administrators were familiar.

Were it not for the rapid development and deployment potential of Perl, Perl would not be an obvious choice for CGI scripting. Perl, after all, offers a double performance hit: first the web server must invoke the Perl script, which means creating a new process, then the Perl interpreter must compile and run the script. (For this reason, later web servers have built Perl interpreters into them. Look at Apache's `mod_perl` and the ISAPI DLL version of Perl from ActiveState.) But Perl could do everything C could do, and, moreover, Perl had built-in text-processing features that made it ideally suited to CGI. Code could be written extremely quickly, since the Perl code could focus on the task at hand without requiring laborious memory-management code necessary for data structures in C. Even though Perl programs did not run faster, they could be *written* faster.

> Perl as a CGI language presents a study in optimization. Most of the time, optimization is performed prematurely and even unnecessarily. Too much emphasis is on optimizing code when there are other ways to correct the problem. Instead of using a more efficient language, vendors and web developers solved the problem of Perl's inefficiency a different way: by embedding a Perl interpreter into the web server itself.

Since the Internet operates in accelerated "web time,"[1] Perl quickly became the de facto CGI language. Perl's position as the traditional CGI language has strong parallels to C's position as the traditional operating system API language. In both cases, a standard arose because developers and employers had to standardize skills on something. It is no accident that every successful language[2] since C has had C syntax! C, C++, Java, JavaScript, and Perl are arguably the most widely used languages today, and all are based on C syntax. Since developers don't want to learn more languages and syntax, and employers want standard skills, Perl took over as the main CGI language.

Perl is the ideal way to keep the porting options open for CGI programs. By this assertion, I mean that you do not tie yourself to any particular web server. A Perl

program is trivial to move to a new platform when compared to, say, a custom-designed ISAPI DLL written in C. Perl's universal implementation has given CGI significant longevity in the fast-changing web world.

Windows, of course, presents a challenge to adapting CGI to work the same way as it does in UNIX. Win32 doesn't have the same kind of process model and doesn't support I/O redirection the way UNIX does. Also, starting a process in Win32 is significantly more resource intensive than starting a process in UNIX. Several alternatives were developed, such as proprietary in-process DLL extensions like ISAPI and NSAPI. Perl didn't really become a feasible language for Windows CGI until the ISAPI DLL version became available.

About IIS and PWS

Microsoft's web server is Internet Information Server (IIS), which comes with and only runs on Windows NT. Microsoft has taken this same server and scaled it down to run on Windows 9x. The Windows 9x version is called Personal Web Server (PWS). The fact is, both IIS and PWS use essentially the same code base. When it comes to discussing CGI, IIS and PWS are equivalent in how they function. The main difference between them is in performance: PWS can't handle the load IIS can. PWS is designed for offline testing and small, low-traffic intranets. For the purposes of this chapter, IIS and PWS are synonyms.

This chapter discusses IIS and PWS only. Many other web servers are available for Windows, but IIS is a common reference platform. Apache for Windows is similar to UNIX. For other web servers, see the documentation that came with the server for how to use CGI.

Web *browsers* are incapable of running CGI programs. Only a web *server* can do the job. Sometimes the uninitiated have trouble with this distinction. Although most modern web browsers are wondrously feature rich and can perform such feats as interpreting client-side scripts, CGI must by its very nature be run on a web server. Fortunately, web servers for offline development are easy to get (both NT and 98 come with then, and so do content-development tools such as FrontPage). It is easy to set up a stand-alone test environment for web development that doesn't require an Internet connection.

Installing Perl on IIS/PWS

My advice is to use the ActiveState ISAPI DLL for Perl. The ActiveState Perl install procedure will configure your computer for you. After installing it, you should be ready to run Perl CGI programs.

You may have to change the IIS registry entry if you *upgrade* Perl. I upgraded to build 504 in the course of writing this book, and changed the Perl directory in the

process. Earlier builds of Perl 5.002 used a different binaries directory than build 504 does.

The association of `.pl` with the Perl IIS component is made in this Registry entry:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\W3Sv
c\Parameters\Script Map
```

The key is `.pl`, and the value will look something like: `C:\Perl\bin\PerliS.dll`. If you ever upgrade Perl and your CGI programs don't seem to work, this entry is probably what you should check first.

Be aware that any time you change the script map section, you must stop and restart the web server for it to pick up on the changes!

Once you've installed the ISAPI DLL, you will have to place your scripts in a directory for which execute permission has been granted. How to set up this directory depends on the web server and the version. By default, both IIS and PWS use `c:\WEBSHARE\SCRIPTS` (or wherever your `webshare` directory has been installed) as an executable directory. The URL for such a script in that default directory would be `http://hostname/scripts/script.pl`. The key is the script directory must be *executable*. Without execute permission, the script won't run. Permission must be set using the web server's own administration tool, not the file system's file permissions. (Even though both the NTFS file system and the web server's administration tool have an "execute" permission, two permissions aren't the same.)

Writing Perl for CGI

Using Perl for CGI programs is natural. Perl was designed as a text processing language, and that is all a CGI program is. CGI programs primarily just process text, both as input and output. All CGI programs are essentially the same. The program reads the standard input, decodes it, splits the `KEY=VALUE` pairs into a data structure such as Perl's hash, processes it, and then prints an HTML page as the output. All steps along the way involve the kind of text processing and rearranging at which Perl excels. During the processing step, the program generally needs to interact with the operating system or another application in order to do something interesting. This C-like ability to access things such as third-party libraries and operating system calls distinguishes Perl from other scripting languages.

Debugging Strategies

If you're new to CGI, one of the most frustrating things about writing CGI programs is simply the fact that they run in an impenetrable black box. If something crashes or

doesn’t work, you don’t even get meaningful error messages generated by the program. All you get are web server errors!

The 500 server error (or any of its close relations) is scary: when you see it, you know something went wrong, but you have absolutely no idea what failed or how to fix it. Most web servers print a useless message about contacting the system administrator if the problem persists. You would think , now that most web servers are several generations old, that web servers would come with some kind of debug mode for CGI programs. Alas, I have never heard of one that does. Most of the for-profit companies that would be expected to include such a thing tend to have their own proprietary alternatives to CGI they would prefer you use. Thus they have no real incentive to upgrade their CGI support.

Solving this kind of problem requires a systematic approach. Either the Perl program did not compile and run successfully, or the program ran and either crashed or produced some output other than what the web server was expecting. First, make sure your CGI program is a valid Perl program. Run `perl -c` on your CGI program at the command line. Next, if your program is valid and you still get the server error, your program may be crashing for some reason. (A valid Perl program that compiles can still crash the interpreter during runtime in some situations, such as when the program can't find a module loaded at runtime.) Alternately, the program itself may either be abnormally terminating or continuing after an unhandled error. The goal is to find a way to get the program to give you output.

The thing to do is force the CGI program to produce valid output so the web server will accept it. If you are trying to debug a Perl CGI program, place the following line before anything that can go wrong:

```
print “Content-type: text/html\n\n<pre>”;
```

(You could also put this line in a `BEGIN` block, but I have always located it near the start of the CGI output.) This line will convince the web server that the output of the program, even an error message, is valid. Instead of a 500 server error, you should see the output from the program.

If your program hangs, that is, if the web browser just sits there trying to download the output for what seems like forever, then you’ve probably used `system()` or tried to open a pipe to or from an external program. I have never gotten this approach to work with IIS. Something in the interaction between the Web server, Perl interpreter, and external program doesn’t work and the program just hangs.

If the web browser downloads and displays the Perl source code *from the web server,* the mapping between the `.pl` file extension and the ISAPI DLL has not been properly configured. If you’ve configured the extension properly, stop and restart the web server. If that doesn’t work, reinstall Perl.

Once you can get the script to run without crashing, and you need to debug it further, remember it can send you e-mail. Store a debug trace in an array and e-mail it to yourself when the program finishes running.

A true story about CGI debugging comes from the early days of SoftBase Systems' presence on the web. At the time, 1994, we had a dial-up UNIX shell account and our web presence hosted on a hosting service. Back then, I had little CGI experience and was trying to write my first program. To make a long story short, I never could get a CGI program to work. I had it working on the command line, and I simplified it to the point where it was only printing `hello, world`, and yet when I uploaded it and tried to run it, I got a 500 server error. After months of trying everything and searching for clues, I finally discovered they had one version of Perl at the shell account, and a different version on the web server. (I finally FTPed the binary of the Perl version on the web server to check it out.) Perl 4 allows e-mail addresses in double-quote strings (`user@host.com`) while Perl 5 forces the @ to be escaped (`user\@host.com`). Therefore, my script worked with the Perl in the dial-up shell but failed on the web server! It drove me crazy trying to figure that out, though, since there was no way to debug it.

Sample CGI Script

I do not want to try to show you what entire books have tackled, but I have included a sample CGI program (see Listing 8.1). It is not particularly fancy.

Listing 8.1 Listing 8.1 A Sample CGI Script

```
#!/usr/local/bin/perl


############################################################
############
# This subroutine is called after the form response has
been picked
# apart and decoded. In it, you should put all of the
output any any
# other processing you plan to do.
############################################################
############
# All form variables are in the associative array %input.
```

```
You access
# them using $input{"value"}.
############################################################
############

use smtpmail;

sub user_output {

print "Content-type: text/html\n\n";

print "<html>";
print "<head>

<script>
function showname() {
  alert(\"My name is $input{name}\");
}
</script>

<title>Hello, Perl World Download</title>

</head>

<body>

<h1>Hello, Perl World</h1>

<p> Sample CGI program's output

<p> <hr>Here is a sample of generating JavaScript on the
fly:
```

```
     <p> <input type=button
         value=\"What's My Name\" name=good
onclick=showname();>


</body> </html>";


$to = 'scott@softbase.com';
$subj = 'CGI output';


@msg = ("This is a test\n", "of the CGI stuff");


smtpmail::smtpmail($to, $subj, @msg);


} # end of subroutine user_output


############################################################
############
# End of user configuration section.
# Do not touch anything below this section, unless you
know
# what you are doing.


############################################################
############
# ungarble: removes all the =20 and %YZ junk from text
passed to the
# CGI program by the browser. It is necessary to do this
because web
# browsers have to encode spaces and other special
symbols in order to
# transmit across the net.  This code is based on a
routine in the
# book CGI Programming in C and Perl.
############################################################
############
```

```
sub ungarble {

  local($s) = $_[0];
  local ($pos, $ascii);

  # replace + with space
  $s =~ s/\+/ /g;

  # replace %YZ hex escapes with ASCII
  $pos = 0;

  while ( ($pos = index($s, "%", $pos)) != -1 ) {
    $ascii = hex(substr($s, $pos+1, 2));
    substr($s, $pos, 3) = pack("c", $ascii);
  }

  return $s;
}


############################################################
############
# Main program: The main program in all CGI programs is
essentially
# the same. It reads the text sent to the program via
standard
# input (I never use the URL-based "get" method), unpacks
it
# into manageable chunks, and then decides what to do
based on the
# decoded input.
############################################################
############


############################################################
############
```

```
# Break apart the information we got from the browser
##########################################################
############

# Since the browser and the web server are nice enough to
tell us how
# much data they're sending, we can get the
CONTENT_LENGTH and read it
# all at once. We don't have to loop through successive
lines or any
# of that business.

# This code handles either GET or POST requests. Code for
this was
# contributed by Jim Lawless.

if ($ENV{"REQUEST_METHOD"} eq "GET") {
  $x=$ENV{"QUERY_STRING"};
}
else {
  read(STDIN, $x, $ENV{"CONTENT_LENGTH"});
}

# What we get is a tightly packed glob of text. Inside it
are
# name/value pairs, each of which is separated from all
the others by
# an amperstand (&). The first thing we do to unpack the
data is to
# split each of the name/value pairs out from the glob
into its own
# slot in an array (named nvpairs).

@nvpairs = split(/\&/, $x);

# Now that we have all the name/value pairs in an array,
```

```
we have to
# split them into their respective halves. Notice how I
use the
# extremely idiomatic for construct here. It's important
to learn this
# Perl construct, since iterating over arrays and
associative arrays
# is so common and I do it so much I use the shortest,
simplest syntax
# possible to reduce clutter.  "for" is actually short
for "foreach',
# but there's no good reason to use "foreach" when "for"
will do. Perl
# emulates the C for loop as well, but the interpreter
can figure out
# which one you mean, and I never use the C-ish for loop
anyway. I
# don't specify a temporary variable name, since it
defaults to $_,
# the current line. That's good enough, particularly in a
CGI program
# where we never have to deal with the current line from
STDIN
# anyway. The expression in parens in the array to
iterate over. The
# loop body will be executed for all members of the
array. (Note that
# I often use "keys %whatever", which gives me a regular
array of all
# the keys in an associative array. That's also a very
common Perl
# idiom.

for (@nvpairs) {

  # Each name/value pair is separated by an equal sign
(=),
  # so we can easily split it into two parts using the
  # split operator. Notice split defaults to the current
```

```
  # line, which is the current array index we're
iterating
  # over using for. That's one reason I like for's
defaults.
  # It makes the code cleaner because you don't use a
million
  # temporary variables.
  ($name, $val) = split(/=/);


  # We have to ungarble the input to get rid of funny
MIME
  # characters.
  $name = ungarble($name);
  $val = ungarble($val);


  # Now we're going to create an associative array of the
  # name/value pairs so it will be easy to access them
later
  # on in the script. Each name/value pair in the input
  # goes into this associative array.
  $input{$name} = $val;


}


# All this has been overhead to this point. Now, call the
real code
# that does the world.


&user_output();
```

Interestingly, Randal Schwartz suggested using a "here" document instead of a long, quoted string with embedded newlines. Since I come to Perl from a C background and have never done any extensive shell programming of any sort (the last shell programming I did was in 1992, before I learned Perl), this possibility would never have even occurred to me.

Next, here is a sample HTML page that will collect data and submit it to this CGI

program. Note that if you use this page, you will have to change the action in the `<form>` tag to match your web server's URL.

```
<html>

<head>
<title>Example CGI Program</title>
<meta name="GENERATOR" content="Microsoft FrontPage 3.0">
</head>

<body>

<h1>Example CGI Program</h1>

<form method="POST"
action="http://scotthome/autobook/excgi.pl">
  <p>Enter your name: <input type="text" name="name"
size="20"></p>
  <p><input type="submit" value="Run The Sample"
name="B1"><input type="reset" value="Reset"
  name="B2"></p>
</form>
</body>
</html>
```

CGI programs are essentially all the same: they decode the special characters, unpack the data `POST`ed by the user,[3] and put the user variables in an accessible place. In this program, `ungarble()` removes the MIME encoding, the main program splits out the packed contents read from `STDIN`, and the hash `%input` gets the `KEY=VALUE` pairs submitted by the `POST` method.

I've only written one CGI program, and I have copied it over and over through the years, changing the `user_output` function as needed for various applications. For some reason, 90% of the CGI programs I have written have been designed to collect user data and have almost all been identical about what they accept from the form and how they process the data. My form predates the widespread use and adoption of `CGI.pm`. In fact my form dates back to Perl 4 days when CGI solutions were ad-hoc.

One topic I have rarely seen discussed is how to use a CGI program to generate

JavaScript code on the fly. People often ask how to create dynamic web pages within the CGI program itself and do not realize that the CGI program can output *any* type of HTML, including scripts. By plugging in JavaScript functions that are created on the fly using variables passed into the CGI program, you can achieve some very fancy effects.

You can print JavaScript in the output as easily as you print anything else, as long as you watch out for the JavaScript characters that are also Perl special characters. You can insert the value of Perl variables in the JavaScript code to create custom functions. In the example, I create a function called `showname`:

```
function showname() {
  alert(\"My name is $input{name}\");
}
```

Note that I had to escape the quotes (`\"` and `\"`), and I used the Perl variable `$input{name}` directly in the JavaScript itself to personalize the function.

Why Windows Is Compelling for the Web

Windows is a compelling platform for CGI because a Perl program in Windows can do so much using Automation. In UNIX, about all a Perl program can do is read text files and maybe issue SQL queries to a database. In Windows, a Perl program can get data from word processors, spreadsheets, and software development tools. A Perl script can also cause programs to generate web pages and graphics. Particularly in light-traffic intranets, the total integration of various programs into an intranet is compelling, when documents from heterogeneous sources can be viewed from the same web browser interface. NT has an advantage over UNIX for small, controlled intranets because lets you integrate applications like Excel and Access into web solutions. A solution based on these applications would probably not scale well to a high-volume Internet site.

One of Windows' strengths is as a testing platform. Software development tools in Windows are decades ahead of their primitive UNIX counterparts. If you dial into an ISP to put your web presence online, you can replicate a full web testing environment on your local PC. If you do off-line testing on a Windows machine, you can also crash your machine all you want without affecting mission-critical programs. Windows is also more productive for software development because you can easily run other programs on the same machine, such as web-development tools, word processors, group scheduling products, and other software products that have no real UNIX equivalent. (The availability of UNIX-based end-user software has increased with the widespread

support for Linux by companies such as Corel. The beginnings of a movement to provide end-user software for Linux got underway as I was writing this book, and by the time this book is available it could be serious. At the time this book is being written, software support for end-user applications in UNIX is still primitive, and web development tools like FrontPage, Image Composer, NetStudio, ScriptBuilder, and Paint Shop Pro have no real UNIX equivalent.)

Sometimes you don't get to choose at all, as I found on a recent project where I had to use some specialized software that only ran on Windows. My program had to interface with other software that was available only on Windows, so my program had to run on Windows, too.

The number of prewritten software packages for Windows makes it compelling. Essentially, NT has done for computing what has not been possible since the days of the mainframe: provide a single OS and platform to write to. The so-called "Wintel" standard allows independent software developers a chance to write sophisticated applications that have a better-than-average chance of running on off-the-shelf hardware.

Why Windows Is Not Compelling for the Web

Windows may be a compelling platform for CGI scripts, but Windows still has problems. I feel qualified to discuss why you might not want to use Windows as a web server platform, since I am one of the few people who has used both UNIX and Windows side-by-side for several years now.

Windows NT with IIS, or Windows 9x with PWS, is an extremely compelling *testing and development* platform. I love having my own web server with which I can do anything. I can map directories, run programs, stop and start it, and do anything without affecting my production web server environment.

Windows is not compelling as a *production* platform. I've used both NT and many flavors of UNIX, and I have come to the conclusion that NT is not good as a production platform for these reasons:

- NT is simply not as robust as UNIX. The NT boxes I have maintained always need a periodic reboot. It's as if they forget what they're supposed to be doing. Where I work, our Internet gateway, an NT server that acts as our ISDN connection and as a router, will drop the connection and not restore it until I reboot. It does this about once a month. But I can't ever remember having to reboot my UNIX machines other than when I intentionally had to shut them down or reboot them for maintenance reasons. I just leave them running. NT machines need more care and attention, and for a busy administrator, that's not good.
- NT inherits the Windows "DLL hell" model. If you don't know, DLL hell is what people call the amazingly unplanned, unregulated structure behind DLLs in

Windows. The original Windows was designed to run on machines with a maximum of 1MB of memory, and because of the small scale, no real attention was given to the regulation of system and application DLLs. For a long time, applications used global system configuration files and stored application-specific DLLs in the system DLL directory. Microsoft did not plan or regulate this at all—there was no mechanism for version control or anything. The burden was squarely placed on an individual application not to mess things up, and with the hundreds of thousands of Windows applications available, such a burden is (statistically) impossible to bear. DLL hell really started in earnest when Microsoft began releasing pseudo-operating system upgrades. These were add-ons like the common dialog boxes, OLE 2.0, MFC runtime DLLs, Visual Basic runtime systems, and so forth. Each application was forced to ship all the runtime systems it needed. It is normal to install something on Windows that breaks other components, because old, incompatible, or buggy releases of pseudo-systems clobber working systems. It is not infrequent or unusual to have to reinstall applications when they get broken because of other applications being installed. It seems to me that NT needs to be rebooted if you even look at some settings, let alone change them. NT takes *forever* to reboot. Certainly, the • number of reboots required even for minimal maintenance on an NT computer is excessive, so much so that the rebooting alone might disqualify NT as a production server. Microsoft has said they want to change this in the future, but I don't see how they can. NT's "DLL hell" architecture doesn't let this happen. Something would fundamentally have to change in how NT manages DLLs to reduce rebooting. It would be an engineering marvel to devise a system that would work with existing programs.[4]

- When things are going well, NT is superb and easy to use. When things go badly with NT, the result is usually *unsolvable* problems. UNIX doesn't have this problem: in 25 years, every corner of UNIX has been explored and mapped. It's like having a complete NetHack game mapped out down to the lowest level.[5] NT, by contrast, is so new that it's only on level 3 of a NetHack game. When something is wrong on a UNIX machine, someone somewhere has an idea of how to fix it. When something goes wrong with an NT machine, you might find someone with the same problem, but it is unlikely you'll find an answer. All you can do is wait for upgrades or find workarounds.
- One severely crippling aspect of Windows NT is that you can't remotely log into an NT machine. If something goes wrong, you have to be there at the console to fix it. You can log into a UNIX machine from home. If you're the network administrator, and something goes wrong on Friday night and they call you at home, which do you want to do? Go back to the machine room and fix the NT machine, or log into your UNIX machine from home? NT's single-user-at-the-console mode is a tremendous weakness, especially for a small business where there can't be an administrator on

site 24x7. (Granted, there are some remote-control software options out there, but why pay an extra hundred and some dollars for something that should come with the operating system?)

- Although the GUI administration aspects of NT are highly touted, and Microsoft would have you believe that Windows NT is easier to operate than UNIX. I've used both and I maintain that NT is no less cryptic than UNIX. Sure, there are some easy-to-use graphical tools, but they only cover some common cases. If you stray off the beaten path, you immediately wind up in no man's land. NT also has some bizarrely arcane ways of doing things. To set up routing over a RAS dial-up, which is probably what 90% of small-business NT users will be doing, you have to make weird Registry changes that are obscurely documented only in Microsoft's tech support Knowledge Base. And logging in a default user • on bootup is another set of Registry modifications that are obscure. You'll encounter things like this every time you try to do something.

In summary, Windows' biggest weakness as a web server platform is its lack of maturity. In a test environment, this frailty is generally no big deal. But in a production environment where the server is expected to be up and running constantly, delays and downtime caused by scrambling to find workarounds for Windows problems are just too costly.

Notes

1. A traditional pre-web software pckage might get a new release each calendar year, so a "web Year" became a web browser (particularly Netscape's) release cycle, which was much faster than a calendar year. Even this cycle wasn't fast enough, as software went into continuous beta releases without ever seeming to be finished.
2. As I said in another chapter, the only exception I can think of is Visual Basic.
3. I never use the `GET` method in a CGI program because `GET` has an intrinsic limit to the number of characters that can be passed to it; a complex form may easily overflow that limit. I have used GET in other situations to pass short parameters as part of a URL, where the complexities of POST were not worth it. Randall Schwartz comments: "No--use GET when the operation can be repeated without messing things up (that way, it can be cached by proxies). Use POST when it changes things on the server (and therefore shouldn't be cached). That's the rule. Please suggest it, and follow it yourself." Since I have never been in a situation where I used a proxy server, I have absolutely no idea what he is talking about. I've always either been on the entire Internet without a proxy server or on a local intranet.
4. To devise such a system would not be impossible, however; Windows 9x is an engineering marvel the way it supports Win32, Win16, and DOS at the same time and is as backwards compatible as it is. If they can pull this off, they can do anything.
5. If you don't know what NetHack is, it's an old VT100-style terminal game where your character was an @ on a character-mode screen exploring level after level of randomly created dungeon. If you've never played it, you ought to, since it is part of hacker lore. A Win32 version is available. See `http://pinky.wtower.com/nethack` for more information.

Chapter 9

The TCD95 Command

In this chapter I take a little trip back to the days of DOS and recreate a Win32 version of a forgotten utility called TCD. Along the way, I discuss the use of Perl within the command prompt, an area of frequent confusion.

TCD95 is an example of a different kind of Windows automation: instead of performing a task automatically in the background, TCD95 automates a daily task (navigating around the directory tree from the command prompt) so you will type less. Commands like TCD95 can be time savers if you use a command prompt much during the day. I have seen very few productivity tools like TCD95 for the Win32 command prompt, which surprises me since these tools used to be quite common in DOS days. If nothing else, perhaps TCD95 will spark creativity in other hackers to write other command-line tools.

> TCD95 is not designed to be used from within batch files and other automated procedures, and in fact, if TCD95 is used in such a way, it presents a serious example of non-deterministic behavior. TCD95 changes the working directory to the first matching directory it finds, and the directory it finds can change if new directories are added. This could break batch files. Always use full and absolute paths in batch files to insure they will exhibit the same behavior every time they are run.

TCD: The First Generation

A long time ago, a gem of a book called *DOS Batch File Power Tools* came out with a lot of nifty little DOS add-ons and commands. One of the niftiest was the TCD command. The TCD command worked like the CD command in the shell, only it searched the directory tree for your directory. You only had to type the directory name, not the whole path. For example:

```
C:> cd c:\Program Files\Common Files\Microsoft
Shared\MSinfo
C:> tcd MSinfo
```

In this chapter, I will develop a Perl rewrite of the old TCD command. This program demonstrates both recursing through the directory tree and using external batch files to set environment variables.

The Environment of the Caller

What is an *environment*? A program's environment is a block of memory composed of `KEY=VALUE` pairs.[1] The `KEY`s are called environment variables, and the `VALUE`s are their values. These pairs can contain almost anything within the limits of the operating system. Some pairs are special to the operating system, such as the `PATH`, but the possibilities for user-defined environment variables are endless.

Where does your program get its environment? From the process that starts the program. Typically, but not always, this process is a shell of some sort. A web server is a good example of a non-shell process that communicates a volume of information to the processes it starts through the environment.

In UNIX, a process environment is simply an exact copy of the environment of the process that starts it, called the *parent* process in UNIX. If a parent wishes the process it starts to have an environment different from its own, it must change its own environment before starting the process.[2] In Windows, a new process gets its environment directly from the `CreateProcess()` API call. The original process has full control over the environment of the new process, because it creates the new process from scratch (like UNIX, typically by copying the existing environment).

The environment has become a generic mechanism for passing data into an application.

The TCD95 command's use of a batch file controller and a Perl program shows one solution to the age-old problem of how a called program can affect the environment of its caller. This book would not be complete without a treatment of this issue.

The general problem is how any called program can modify the environment of its caller. This problem surfaces surprisingly often in scripting situations, particularly when the `PATH` of a calling shell needs to be modified by a called Perl program. It would be nice to have changes to `%ENV` somehow "propagate upward" to calling processes, but it is not going to happen automatically.

By itself, a called process can't modify the environment of the caller. The called process is executing in its own address space and has a separate copy of the block of environment variables from the caller. This observation is true in both UNIX and Win32.

Two notable exceptions: Of course, in the "good old days" of DOS when all processes ran in the same address space and there was no memory protection at all, a determined programmer could scan memory and find the caller's environment and overwrite it. This option does not exist in modern multitasking operating systems like Win32, where each process has a separate, protected address space.

A batch file does not run as a separate process. It is interpreted by the current command interpreter. Therefore, it can modify the current environment. TCD95 takes advantage of this fact. (UNIX shells do not work this way: Normally, shell scripts run in new processes. But they have the . and `source` commands,[3] which execute the script in the current process and thus can achieve a similar effect.)

The only way a called program can modify the environment of its caller is if the two cooperate. In TCD95, the controlling batch file cooperates with the underlying Perl program. The Perl program creates a temporary batch file that the controlling batch file calls after the Perl program terminates (and then the controller also deletes the temporary file). In the case of TCD95, the goal is to change the current working directory for the command interpreter, which can only be changed using the CD command.

Code

Listing 9.1 shows the TCD95 program. A sample batch file and some notes on the code follow the listing.

Listing 9.1 TCD95

```
# TCD95

# Usage: tcd <directory>

# This program consists of tcd.bat and tcd95.pl. Put both in a
# directory on your PATH. You may need to edit tcd.bat to get
it
# to work on your system.

# Background:

# I found a copy of DOS 5 Complete by Manfred & Helmut
Tornsdorf at
# a book liquidation, and grabbed it. I enjoyed the depth of
technical
# detail, batch file and code samples, and completeness. I
discovered
```

```
# the authors had written a book called Batch File Power Tools
as well,
# but it took months to locate it. When I did, it became one of
my
# favorite DOS books. One gem included was the "tcd" command,
which
# basically was the lazy programmer's dream command: it was
like the
# CD command, but it travered into subdirectories looking for
your
# directory. Instead of typing "cd \docs\letters\short\p1995",
you
# could type "tcd 1995" and the command would search until it
found
# the directory. This was slow, obviously, but these were
the days
# of DOS (the program copyright date is 1990!) before command
# completion existed -- even with completion, remembering where
in
# a deep tree a particular directory was is sometimes hard.
#
# The biggest drawback to tcd on a Windows 95 box is simply
that it
# gags and dies on long file names. It'll work with shortened
# filenames, but that's not too useful.
#
# With directory names like
#
#        "c:\Program Files\Common Files\Microsoft
Shared\MSinfo",
#                      and "g:\Delphi 2.0\IMAGES\BUTTONS"
#
# becoming normal on Windows 95, I felt an acute need to bring
this
# program into the modern era.
#
# A couple of drawbacks emerged while I was writing tcd95:
first of a
# all, although the original tcd.exe program changed the
```

```
directory for
# the shell, it isn't that simple in Windows 95. The tcd95.pl
program
# has to write a batch file that the command processor then
calls,
# because changing the directory in the child process does not
change
# the directory in the parent process. The other drawback is
the batch
# files deleting the temp when it is done with it -- I put the
temp in
# the root directory so it would be easy to delete no matter
where we
# CDed to.
#
# I wrote this in Perl because it is trivial to write recursive
# directory processing code.
#
# NOTES:
# 1. This version does an inorder traversal of the directory
tree.
#    That means it starts at the root directory and traverses
into
#    each subdirectory as it finds it, and traverses into lower
#    directories before going on to directories at the same
level.
# 2. Traversal is in whatever order readdir returns. This seems
to be
#    the order in which the directories were created on the
disk.
# 3. In the event of multiple directories with the same name,
this
#    program stops at the first one it finds. (In practice I
find this
#    isn't an issue -- I've rarely if ever seen two directories
with
#    the same name on a system.)
# 4. This program only works on the current drive, it does not
attempt
```

```
#    to search other drives. It's slow as mud now, that would
#    essentially cripple your system for days (if you have as
many
#    drives as I do!) HINT: the exit code of the program if the
#    directory is found is 5, but it is 4 if it is not found.
#    Therefore, someone who wanted the feature badly enough
could hack
#    the tcd.bat file to search any number of drives.


# Wish List
# [ ] Specify inorder or level-by-level traversal
# [ ] Specify directory sort order so you can search most/least
recent,
#     alphabetical, etc
# [ ] Case (in)sensitivity switches
# [ ] A tcd environment varible with default options for all
this stuff
# [ ] check for invalid filename or non-quoted long file names


if (defined $ARGV[0]) {
    $d = $ARGV[0];
    print "Attempting to change to directory $d (be
patient)...\n";
}
else {
    print "usage: $0 <dir>";
}


# spannerize is our directory tree walker
spannerize("/");


# if this function returns, we're in trouble
print "I give up -- $d doesn't seem to exist on this drive.\n";


# not found, so return 4
exit(4);
```

```
################################################################
######
# spannerize: span directory tree
# The name of this function is a play on "span", i.e. span the
# entire directory tree, and a play on "spanner", i.e. a tool
################################################################
######


sub spannerize {

    local($startdir) = @_; # local var because this is
recursive

    opendir X, $startdir or die "Can't open $startdir, $!\n";

    # we grep for everything but . and .. which would cause
    # infinite recursion
    for $x (grep !/^\.\.?$/, readdir X) {

                    #we only want directories
                    if ($startdir eq "/") {
                        if (! -d "$startdir$x") {
                           next;
                        }
                    }
                    else {
                        if (! -d "$startdir/$x") {
                           next;
                }
            }

                    #does the name match?
            if ($x eq $d) {
                $startdir = "" if $startdir eq "/";
```

```
                print "Directory is now [$startdir/$x]\n";
#               chdir("$startdir/$x") or die "What the--!?\n";
#               system("cd $startdir/$x");
                open(T, ">/tcdtmp__.bat") or die "Internal
error: $!\n";
                print T "cd $startdir/$x\n";
                close(T);
                exit(5); # good exit value
            }
            # if not, traverse deeper
            else {
                if ($startdir eq "/") {
                          spannerize("$startdir$x");
                }
                else {
                          spannerize("$startdir/$x");
                }
            }
    }
}
```

This program is not useful unless it is combined with this batch file:

```
@echo off
: This batch file is the driver for tcd95.pl. Please see
the comments
: in that file for more information on what this command
is and how
: it is supposed to work.
perl tcd95.pl %1
call \tcdtmp__.bat
del \tcdtmp__.bat > nul
```

Code Notes

To use this command, type `TCD` followed by the directory to which you wish to change

at the command prompt:

```
d:\> tcd autobook
d:\scott\work5\autobook> _
```

If you've configured your command prompt to show the path as part of the prompt, the change to the new directory will be immediately reflected.

`TCD.BAT` is a batch file driver for `tcd95.pl`. The batch file calls the Perl program, and the Perl program generates a batch file. The batch file then calls the generated batch file to change the working directory. Once the batch file has been called, it is deleted.

The function `spannerize` is called recursively starting at a specified directory, in this case the root. I purposefully wrote `tcd95` to be able to start at any arbitrary directory in the tree, since I knew the code would be useful in the future (as it was in `tare`, a command introduced in the next section based on this code).

In any recursive directory function like this, the special cases of `.` and `..`, two directory names returned as part of the `readdir()` call, must be handled: if they are not skipped, the recursion will go on forever.

The `tare` Command

Similar to the `TCD` command is `tare`,[4] another directory-recursing program I wrote for deleting all the files of a certain type in a directory (see Listing 9.2). `tare` is the kind of ad-hoc, situational script that I love to write in Perl. It was written quickly to fill a specific need.

The origin of the script came soon after I began doing backups to a Zip drive rather than to floppies. With the spaciousness of these drives, I began to put all the projects I was working on with various development tools in one directory. I would then compress the whole directory and copy the archive file to the Zip drive. The problem these days is not lack of space but the inordinate time it takes to do a backup (especially if you have a single Zip drive that's shared on the network and you access it from other machines).

Visual C++, like most modern development tools, trades cheap and plentiful disk space to save compilation time. Precompiled headers, incremental linking, and browser info databases save time and also create huge files. The only problem with this approach is that when you start to do backups, it's hard to pick out the good files from the unnecessary ones.

Luckily, if you delete most of these space-consuming files, development environments such as Visual C++ regenerate these files the next time you rebuild

your project. Deleting the files for backup purposes is not much of a problem. You might work on a project all day and want to back it up at the end of the day before you turn off the computer, in which case you'll only have to rebuild the files once the next morning when you rebuild the project.

Perl is an ideal choice to handle this chore, especially if you have `TCD95` already written. Both are based on the concept of recursing a directory tree. The code looks a lot like `TCD95`. At any rate, I don't think it is possible to write this type of recursive processing in a batch file.

Listing 9.2 `tare`

```
# tare: weed out files from the directory tree

@deletia = ("*.pch", "*.bsc", "*.obj", "~*.tmp", "*~");

if (defined $ARGV[0]) {
    $d = $ARGV[0];
}
else {
    print "usage: $0 <dir>";
}



print "Weeding...\n";

# spannerize is our directory tree walker
spannerize(".");

print "Done!\n";

exit(0);

############################################################
############
```

```
# spannerize: span directory tree
# The name of this function is a play on "span", i.e.
span the
# entire directory tree, and a play on "spanner", i.e. a
tool
############################################################
############


sub spannerize {


  local($startdir) = @_; # local var because this is
recursive


  print "Directory is now [$startdir], deleting files\n";


  for $f (@deletia) {
    $delcmd = "command /c del \"$startdir\\$f\"";


    print "Delete command: $delcmd\n";


    system($delcmd);
    print "\n";
  }
  print "Done with $startdir...\n";


  opendir X, $startdir or die "Can't open $startdir,
$!\n";


  # we grep for everything but . and .. which would cause
  # infinite recursion


  for $x (grep !/^\.\.?$/, readdir X) {


    #we only want directories
    if ($startdir eq "/") {
```

```
      if (! -d "$startdir$x") {
                  next;
      }
    }
    else {
      if (! -d "$startdir/$x") {
                  next;
      }
    }

    keep_going: {
                  if ($startdir eq "/") {
                     spannerize("$startdir$x");
                   }
                  else {
                     spannerize("$startdir\\$x");
                   }
     }
 }
}
```

The main difference between `tare` and `TCD` is that, in `tare`, I run a set of delete commands in the current directory and in each directory under the current directory.

The list `@deletia` contains shell wildcard patterns for files that I wish to delete. For each directory encountered (starting with the directory specified as a command line argument), `$delcmd` will be run via the `system()` function for each pattern in the `@deletia` list.

tare Redux with `File::Find`

`tare`, which is based on the code in `TCD95`, predates the widespread and easy availability of Perl modules on Win32 systems that came with the grand unification of the Windows and UNIX Perls in version 5.005. `tare`, therefore, does not use the Perl library module `File::Find`. Before Perl 5.005, I never used library modules and have always been suspicious of them. For one thing, you can't ever count on every library module being available on a given system, so programming with them is perilous if you have the need to move your scripts around. (I learned that when I did an eleventh hour job using Perl to download some URLs and un-HTML-ify the HTML code to produce plain text. The program worked great on Windows 98 but would not run on UNIX because the `LWP` module was nowhere to be found. It was too late to figure out how to get `LWP` on the UNIX box, and I wound up running the script on a Windows 98 box.) If you are willing to risk using them, though, modules can save you considerable time by providing high-level bundles of functionality. Modules are Perl's answer to Automation and standard libraries like the C++ and Java runtime standard libraries, and they pack a punch.

Listing 9.3 shows the `tare` command (developed in the last section) using the `File::Find` module to demonstrate modules in action. The module-driven program is somewhat similar to the non-module `tare` but is almost a mirror image of it.

Two big changes have been made to the code:

- Instead of going into each directory and deleting a shell wildcard pattern, each file is processed (since that's the way `File::Find` works) to see if it matches any of a list of patterns. This scenario is inside-out from the previous implementation, which processes a list of patterns one at a time per directory.
- Patters are *regular expressions* rather than shell wildcards. This fact represents a certain danger, since regular expressions must be handled with more care than wildcards. Regular expressions are much more precise than the comparatively vague wildcard patterns. Among other things, you must be careful to escape all literal periods in the filename (since a period is a regular expression character); you must not use double-quotes to store the regular expression in a Perl string or bizarre side-effects will happen as Perl tries to perform variable substitution on the dollar signs (use the safer single-quote instead); and you must be careful to specify the end of string `$` for filename matches containing extensions. The last one • in the list is a gotcha: the *shell* assumes the end of the string has an implied `$`, so `*.obj` will match `foobar.obj` but will not match `foobar.obj.old`. The *Perl* regular expression match has no such guarantee and `\.*.obj` will happily match

`foobar.obj.old`, while `\.*.obj$` will not. It's easy to create a disaster if you are using regular expressions to delete files (which is what `tare` is doing), since regular expressions require an attention to detail the shell does not require.

For this program and any other program that deletes files, I strongly recommend you test first by printing out what you are going to delete instead of actually deleting it. That way you will know if the regular expression (or even the shell wildcard) will do what you intend or not without suffering the consequences. (Remember, the Army doesn't let people have live ammunition at first, either.)

The warning "Always make a backup before doing this" is so common that it has lost most of its urgency. It is heard but never listened to. I am of the opinion that all the talk in the universe is insufficient to motivate people to create backups. The best motivation for making backups is to lose something you really don't want to do over, and after that the problem takes care of itself. You'll either become backup-paranoid like I am or never learn. Nevertheless, I feel compelled to warn you that playing with a command like tare, which can delete a large number of files automatically, warrants extreme caution. You should definitely make a backup, and you may want to change the code to just print out the commands before actually running them.

Listing 9.3 `tarefind.pl`

```
##############################################################
############
# This is tarefind.pl, tare.pl rewritten to use the
File::Find module
# instead of manually traversing the file tree. The main
difference is
# this is "inside out" from the other in terms of how it
processes the
# files. Tare loops, processing each file inside a loop.
This writes
# the processing as a sub, and passes that to File::Find,
which does
# the recursion for you.
##############################################################
############
```

```
use File::Find;


############################################################
############
# The biggest difference here is @deletia contains a list
of Perl
# regular expressions rather than shell wildcards.
Regular expressions
# must be much, much more precise. You can easily write
one that will
# delete the wrong thing.
#
# Note:
#
# 1. Enclose the regexps in single-quotes, not double, to
make
# life easier.
#
# 2. Backwhack all literal periods
#
# 3. Put a $ at the end of the regexps, since long
filenames could
# easily look like 'file.extension.old' and you could
delete the wrong
# files.
############################################################
############


@deletia = ('\.pch$', '\.bsc$', '\.obj$', '~.*\.tmp$',
'.*~$');


$dir = "d:/scott/work5";


############################################################
############
# Define a callback. This subroutine will be called each
```

```
time
# File::Find sees a file in the filesystem.
############################################################
#############

sub callback {

  my $f = $File::Find::name;

  #print "Callback called on $f";

  for $pat (@deletia) {
    if ($f =~ m!$pat!) {
      #print ": MATCHED!!! $pat";
      unlink $pat;
    }

  }

  print "\n";

}

############################################################
############
# The main program just starts the find subroutine.
############################################################
############

find(\&callback, ($dir)
```

Notes

1. The environment's ancestor is the DD statement in mainframe Job Control Language (JCL). The DD statement came from a need to make the same program work with different files without recompiling the

program. Prior to the introduction of the DD statement, a program had to be hard-wired to use certain files, which was too inflexible. Remember that the idea of command line parameters did not yet exist. The command line didn't even exist. The idea of a DD statement was built into JCL and the I/O routines for various language compilers. To make a program operate on (or produce) a different file, only the plain text JCL file that ran the program had to be changed, not the compiled program. UNIX generalized this design into environment variables, which were much more general in how they could be used than DD names. (Similarly, the UNIX shell concept is more general than JCL.) Environment variables have propagated to all major computer platforms, including Win32.

2.This description oversimplifies the issue. For more details, consult a book on UNIX systems programming.

3. Yes, that first one is a single dot. There's also a command whose name is [. You can see where Perl gets things like $[.

4. Tare is an archaic word for "weed," chosen because it was unlikely to be the name of an existing command.

Chapter 10

Grab Bag

This chapter is a grab bag of topics too short for their own chapters.

One-liners

Perl is great for one-liners, but the Windows command shell makes one-liners hard to write. The Windows shell presents a Catch-22: You can't use double quotes in the code for the one-liner because you have to use double quotes to surround the one-liner on the command line. The solution is to use a curious operator called `qq`, which makes the character following it the string delimiter. If you use a natural "begin" character like `{`, then the natural "end" character `}` is the closing quote. (If you use a regular character, the same character is the end of the quote.)

So, for example, you could use:

```
C:> perl -e "print qq{This is strange looking, but it
works\n}"
C:> perl -e "print qq!This is strange looking, but it
works\n!"
C:> perl -e "sleep(10); print qq{time's up\n}"
```

What Is a Good Editor?

A question frequently asked by Windows users is: "What is a good text editor?" A lot of people are asking this question as Windows becomes a more important software development platform, particularly for programming that doesn't involve a user interface (such as Perl, HTML, Java, etc.). Windows itself has always, even back in the days of DOS `edlin`, been pathetic in terms of the editors it supplies out of the box. To begin with, Notepad and the console mode editor `edit.com` are not good text editors. The `edit` command that comes with Windows can't even handle long file names properly. Try editing a file that ends in .html. WordPad is just a Rich Edit Control example program pressed into service as a system applet. Anyone doing any serious work on a Windows system must get a usable text editor. Windows always has been a ripe platform for third-party editors, but which editors are good?

A text editor is a programmer's single most indispensable tool. A programmer *lives*

in the text editor. I can't imagine a day when I *didn't* use one or more text editors, at work or at home. The editor is so important that hackers and real programmers have spent the last thirty years perfecting the text editor, and they have an interest in making these editors available for free. First, the hackers themselves want the editors on whatever platforms they use. Second, they want other hackers and would-be hackers to have easy access to good text editors (so the other hackers can help them hack). There is a role for commercial editors, particularly in Windows. Most free editors are ports from other platforms and are idiosyncratic with respect to their origins. Native Windows *free* editors generally do not have the features most commercial editors have.

The expression “You get what you pay for” is true more often than not. You may wonder if free editors fall into this category. Sometimes. But in the case of GNU Emacs, nothing could be farther from the truth. It is *the* text editor. If software is the brains of the computer, Emacs is the blood that supplies the oxygen necessary for the brain to function. Emacs is so valuable that you can't put a price tag on it. It's the lifeblood of software development over the past twenty or thirty years. But when you get past organized efforts like the GNU project, the quality drop-off is alarming. Some of the dreck on shovelware CD-ROMs is frightening.

The editors I review here come from my own long computing experience.

- Emacs for Win32: The name "NT Emacs" is not accurate, since this Emacs runs well under Windows 9x, too. GNU Emacs for Win32 is the best text editor you can possibly get for Windows, • commercial, free, or otherwise. It's a one-stop editing and software development environment with support for every programming language in use, as well as HTML coding and other popular text editing tasks. GNU Emacs is not the only free text editor, and may or may not be the best one for you to use. Its biggest advantage (completeness and features) is also its biggest drawback. You also may be put off by having to learn all the keystrokes and all the syntax of what can be an arcane extension language. Don't be. Like Perl, Emacs will return any investment you make in it a hundredfold. Emacs is available on most if not all modern platforms, is amazingly consistent between them, and is one of the most full-featured editors ever created. Its powerful customization language, Emacs LISP, allows you to reinvent the editor to do anything you want. You need to work with Emacs until the keystrokes are wired into your nervous system and you don't even consciously think about them. The biggest drawback to Emacs under Windows is its almost utter lack of printing support. It can generate PostScript files, but printing these PostScript files is a clumsy process. If you're used to just clicking an icon and having pages come out of your printer seconds later, Emacs will be a disappointment. For printing source code, I supplement Emacs with another editor. You can get the Win32

version of Emacs from `http://www.cs.washington.edu/homes/voelker/ntemacs.html`. As this book was going to press, Emacs version 20 became available for the Win32 platform in a very stable release. This editor is, without question in my mind, the best editor ever created.

- Programmer's File Editor: Programmer's File Editor (PFE) is not derived from a UNIX editor. PFE sports a very nice native MDI interface that follows accepted Windows conventions. Unlike most native Windows editors, PFE imposes no arbitrary file size restrictions. You can edit just about anything that will fit into memory. PFE is particularly good for printing files. It gives you full control over the font in which the text will come out (so you can pick your favorite non-proportional font), and allows you to number lines and pages. Another PFE plus is the ability to capture the output of DOS commands into a window. This feature was more of a plus back when Emacs could not run inferior processes in Win32, but the current version of Emacs can now run these processes just like the UNIX version. I recommend keeping a copy of PFE handy even • if you use Emacs, particularly for printing. This editor has a few drawbacks, such as not supporting syntax highlighting. These drawbacks seem minor when you consider that PFE is totally free. To get this software, go to `http://www.lancs.ac.uk/people/cpaap/pfe/`
- MultiEdit 8.0: The best Windows commercial editor I have used is MultiEdit, from American Cybernetics. If you're willing to spend money, MultiEdit is a deluxe luxury editor that has great support for Perl programming. I originally purchased 7.11 for its Delphi integration. With 8.0, they've created an impressive native Win32 editor. MultiEdit is a native Win32 application that follows normal Windows conventions and is completely customizable. In addition to Perl, it supports many other languages including HTML and Delphi. For information on MultiEdit, visit `http://www.amcyber.com/`. (There are certainly many other commercial text editors with equivalent features, but I feel MultiEdit gives you the best value for the money: it is usually over $100 cheaper than the competition, does everything I've ever needed it to do, and has excellent Perl support.)

Note: If you're using Emacs on Win32, you'll probably want to add lines such as these to your `.emacs` file:

```
(define-key global-map [end]    'end-of-line)
(define-key global-map [home]   'beginning-of-line)
(define-key global-map "\C-z"   'undo)
(global-font-lock-mode t)
```

The first three lines remap keys to what Windows users would expect. By default, Emacs does very non-Windows things when these keys are pressed, which can surprise you. Font lock mode is what Emacs calls syntax highlighting, and normally you want font lock mode to be on in every buffer.

Although the editors profiled above are the *best*, here are some alternatives:

- MicroEmacs: MicroEmacs is an offshoot from the Emacs family tree that took a radically different path from GNU Emacs. As a result, the two are just similar enough to trip you up. The keystrokes and conventions this editor uses are similar to, but not the same as, GNU Emacs. The MicroEmacs editor's extension language is radically different from GNU Emacs' Emacs LISP. The Windows port of MicroEmacs is actually • very well done. It manages to combine the best of the old text-mode MicroEmacs with convention-following GUI enhancements. For the most part, I recommend using full-blown GNU Emacs unless there is some specific reason not to. You can download various MicroEmacs binaries from `http://members.xoom.com/uemacs/nojavascript.html`. A different version, the *JASSPA* distribution can be found at the web site `http://www.geocities.com/ResearchTriangle/Thinktank/7109`.
- NotGnu: One of my favorite editors under DOS was mg, a very small clone of the GNU Emacs editing look and feel that fits on a floppy diskette. It's great for emergency repair work. You can still find this at old DOS software archives, and it's good to have around for those emergencies. Mg has survived into the 32-bit world of Windows 95 and NT as NotGnu. It has been extensively GUI-ified, and no longer has a command prompt interface. I haven't used it enough to recommend it over Emacs itself, but it is much smaller if you are tight on memory or disk space. If you use GNU Emacs, you will probably want to track down a copy of the old `mg` for your emergency DOS boot disk. Although NotGnu seems to have disappeared from the face of the Internet, the old DOS binaries of mg can still be downloaded from `ftp://ftp.simtel.net/pub/simtelnet/msdos/editor/mg2a_exe.zip` (source code is `/mg2a_src.zip`).

I do not have any specific recommendations for vi clones. I've tried several native Windows vi editors, and none of them supported calling an external process, a feature required to support the `!}fmt` idiom, which I use constantly while in vi. The editor just doesn't seem to port well to a GUI. Of all the console mode versions, elvis is definitely the best. Overall, GNU Emacs' vi emulation mode is about the most complete, easiest-to-use vi clone for Windows available now.

I have a mini-review of Windows text editors on my web site, from which this section originated, and I occasionally get e-mail from people who wonder why their favorite shareware editor is not listed. I do not have any recommendations on shareware

editors because I've never seen one that stacks up against commercial and free editors. I have not found a shareware editor that offers enough added value over free editors like PFE to make it worth paying for. Likewise, the commercial editors give so much more power than any shareware editor I've seen that they're worth the extra margin over the price of the shareware editor. Shareware editors occupy a strange in-between niche between the two worlds.

Perl and the Windows Scripting Host

As I have tried to impress upon you throughout this book, Microsoft's strategy for Automation and scripting is to separate scripting language engines from the objects on which they operate (Automation objects). Microsoft has bundled common system administration tasks into Automation objects that can be called either by scripting languages that run under the Windows Scripting Host (WSH) or by Automation client languages.

WSH is an environment that can run various scripting languages. (JScript and VBScript come with Windows 98 out of the box.) The scripting language must conform to a certain WSH language standard defined in terms of COM. The problem is that WSH is so new that no one really understands it. The Windows 98 help doesn't even have any information on Windows Scripting Host. Windows 98 is the first platform to come with WSH, although WSH can be installed on other Windows platforms. (I have never actually done so, but I have heard it can be done.)

> In a way, I am happy to see some form of BASIC return to Windows. I first started programming using the built-in BASIC on a TRS-80 Color Computer 2 (a.k.a. the CoCo), and programming in that BASIC led me to a career as a programmer. Since Windows 95 replaced DOS, though, the microcomputer BASIC disappeared and nothing has replaced it. I am glad to see WSH bring BASIC back to the PC in the form of VBScript.

WSH scripts have access to some objects that greatly simplify certain actions. The main bundles of functionality are WScript.Shell and WScript.Network, although there are some others. Shell gives you access to all sorts of aspects of the Windows shell, and Network gives you the ability to perform network-related tasks like mapping drives and seeing what connections are present on the system.

These objects can be called from regular Perl programs.[1] The WSH objects are just Automation objects and can be created using the `Win32::OLE` module (as shown in Listing 10.1).

Listing 10.1 Windows Scripting Host Example

```
use Win32::OLE;

# Using the shell objects ...
$ws = new Win32::OLE 'WScript.Shell' ||
  die "Couldn't create new instance of WScript.Shell";

# ...you can run programs...
$ws->Run("calc.exe");

# ... access environment variables...
$path = $ws->Environment()->{"PATH"};

# ... and access special variables.
$os = $ws->Environment()->{"OS"};
$comspec = $ws->Environment()->{"COMSPEC"};

print "PATH: $path\n";
print "OS: $os\n";
print "COMSPEC: $comspec\n";
```

I can think of no compelling reason to use PerlScript from the Windows Scripting Host. Regular Perl programs can do anything a PerlScript program running under WSH can do.

Perl and Y2K

I am compelled to say something about the Y2K situation, even though by the time this book comes out the point will be largely moot.

Perl itself, like anything based on the UNIX time library, has no intrinsic problem with the year 2000, since the entire library is based on dates since “the epoch” in 1970.

Amateur astronomers, conscious of time issues, designed the UNIX time library to be extremely accurate and flexible, and it was ratified as-is to become the time library for the ANSI C standard run-time library. Perl just uses the C runtime library as its own time library.

Every date and time manipulation in the C runtime library is done as a `time_t`, an integer number of seconds since the epoch. When converting a time from `time_t` to a human-usable format, the year is represented as the number of years *since 1900*, so it also has no intrinsic year 2000 problem.

The difficulty with `time_t` is simply that it, too, runs out and overflows at some point in the future. With a 32-bit integer, the overflow happens in 2038. The good news is that simply by increasing the size of a `time_t` to 64-bits, you solve the problem for a very long time. By 2038, it is to be hoped that the transition to a word size much larger than even 64 bits will have been accomplished and that computer architectures will have changed so radically that all code will have to be recompiled.[2]

Perl's immunity to Y2K, of course, requires Perl programs to use the date and time libraries properly. This is a stretch. Even if Perl itself is compliant, there's nothing to stop a programmer from taking a shortcut and using, say, a two-digit date in some file format or printout. Programmers must also use `time_t` responsibly, as an abstract data type, and not assume it is equivalent to any particular machine data type.

Why `fork()` Doesn't Exist in Windows

As this book was going to press, a major announcement came out that ActiveState is going to try to add `fork()` support to a future version of Perl This section gives you an appreciation of the isues they will have to address.

UNIX programmers who come to the Win32 operating system usually ask sooner or later why `fork()` is not supported in Perl under Windows. I have never seen a good answer to this question, so I'll give one here.

It is not that Perl itself doesn't support `fork()` but rather that the Win32 architecture itself does not support `fork()`. A C programmer can't use `fork()` in Win32 either. The reason why `fork()` doesn't exist has to do with the process model of VMS. Yes, VMS: The Win32 architecture is more heavily influenced by VMS than it is by any other operating system.[3]

In Win32/VMS, creating a process is like giving birth to a child. In UNIX, creating a process is like baking a cake. UNIX processes are trivial to create, and the OS supports creating processes at a whim. To create a process, you first create an exact copy of your current address space using `fork()`. You can then either use the `exec*()` family of system calls to load a new program or keep executing the same program you were executing before the `fork()`. UNIX shells do the former, and TCP/IP servers do the latter. In Win32, as in VMS, creating a process is a *big* event. It

takes a lot of work and system resources. When you've created a process, you've *done* something. Process creation is therefore reserved for starting a new program. Once a program starts, the program typically adds additional tasks by splitting into multiple threads rather than creating more processes. Compared to the amount of overhead required for creating a new process, creating a thread is trivial.

The problem with porting UNIX TCP/IP servers to Win32 is that the UNIX TCP/IP idiom can't be translated to Win32. In UNIX, a master TCP/IP server process runs. When a new connection comes in on the port where the master process listens, the master process calls `fork()` to create a copy of itself and lets the copy handle the transaction. The master goes back to listening and calls `fork()` for every incoming connection.

The UNIX TCP/IP idiom operates on the assumption that you can make a copy of an address space using `fork()`. Win32 has no way to copy an address space. You can use the `CreateProcess()` API call to start a new process, but there is no way to clone the address space of the current process.

Some programmers say you *can* use `fork()` *if* you use some ported third-party runtime environment. It's true, there are implementations of UNIX-like runtime libraries (such as the NT POSIX-compliant subsystem) that have a `fork()` call. The problem is that if you use these libraries, you must limit yourself to them. You can't have a full-fledged Win32 application with all the attendant rights and privileges and also use the libraries for `fork()`. Plus, if you're using Perl, you must recompile Perl under this library, and in so doing you will likely lose extras such as the `Win32::*` modules.

Thread support is the most exciting new feature of Perl. As this book is being written, experimental support for multithreading has also been added to Perl, and in a release or two support for multithreading is most certainly going to become rock solid. With platform-neutral threading support built into Perl, Perl will be like Java in the sense that it will totally isolate software from the underlying thread API. The problem with writing cross-platform threaded programs now is no two platforms have the same API. The Win32 thread API really exists in two flavors: the raw API provided by the OS and your compiler's API (which makes the runtime library function properly within threads). And the Win32 threads are nothing like POSIX threads. You will be able to write TCP/IP servers using Perl's threads and not have to worry about the process model differences between platforms. You also will not have to port programs that use different thread APIs. With Perl and Java supporting platform-independent threads, the UNIX TCP/IP idiom based on `fork()` will eventually fade from memory.

Using a Windows Database from UNIX

I sometimes hear a question about how to use a Windows database from a Perl program running on a UNIX box. Typically, the asker has an Access (or SQL Server) database on the Windows machine and wants to develop CGI programs on the UNIX machine. I try to answer this question, but there is really no good way to respond.

The general problem is one of trying to swim upstream. Data flows from the server OS to the client OS. To go backwards, from a server OS like UNIX down to a client OS like Windows, is either very hard or very expensive.

> All databases I've ever seen have ODBC drivers for Windows because Windows is the client OS used most of the time to access backend server databases. On the back end, ODBC is rarely used. Embedded SQL is used instead.

One solution is an ODBC driver for the UNIX machine. ODBC drivers for UNIX are very, very expensive (and likely not an alternative since anyone who is attempting this configuration generally is using Access because a UNIX database is unaffordable). ODBC drivers are rarely needed for UNIX machines, and, as a result, they are priced higher because they are specialized. The other alternative is to write a TCP/IP server for the Windows machine that reads SQL statements from a socket, executes them, and prints the results to the socket. This solution is probably the best low-cost alternative.

Even if you could somehow do this, Access is itself not powerful enough for the kind of load that being used as a Web backend database would place on it. Access is simply not powerful enough to be used as a server database.

The `graburl` Function

Many people beg for a short, simple module that reads the contents of a URL into a string. My `geturl()` does just that. You pass `geturl()` the URL to read, and it reads the URL. `geturl()` then returns the URL as a string. Like the `smtpmail()` function, `graburl()` is also packaged as a module. This design is a case of building extremely high level modules out of lower-level ones.

You may also be interested in using `LWP::Simple`, which Randal Schwartz pointed out to me when he read the manuscript:

```
use LWP::Simple;
$contents = get $url;
```

The code I present in this chapter predates `LWP`'s availability on Win32. This code originally used raw socket code (like the `smtpmail()` function does). I rewrote the code in terms of LWP for this book. Although LWP makes it very easy to do what `graburl` does, I left my code in because it's a good example of reinventing the wheel because of brain overload. I'm sure, somewhere, I have some documentation that references `LWP:Simple`, but I often find myself so overloaded that I don't even care: I just use the first thing I find that works. In this case, I used the full `LWP` module.

Listing 10.2 shows the `graburl` module.

Listing 10.2 `graburl` Module

```
package Graburl;
use Exporter;
@EXPORT = qw(download);

use LWP::UserAgent;

sub download {

  ($url) = @_;

  $ua = new LWP::UserAgent;
  $ua->agent("AgentName/0.1 " . $ua->agent);

  my $req = new HTTP::Request
    GET => $url;
  my $res = $ua->request($req);

  if ($res->is_success) {
    return $res->content;
  }
```

```
  else {
    return "Download of $url failed\n";
  }

}
```

Following is a driver program that uses the `graburl` module:

```
use Graburl;

print Graburl::download("http://localhost/default.htm");
```

HTML Report Generation and `write()`

Though not common, it is possible to format reports using `write()` in HTML format. There's nothing you could do with `format/write()` that you could not do with a format string and `sprintf()`, but sometimes the paradigm of the detail lines on a report is the most natural way to express the solution to a problem.

You must make sure you put a space between format strings and HTML tags. The following code does not print a five-position field followed by a break:

```
@<<<<<<br>
$field
```

But this code will:

```
@<<<<< <br>
$field
```

Following is an example:

```
$header =
'<html><head><title>A Directory
Listing</title></head><body><table>';

format STDOUT_TOP =
<th>Filename</th><th>File Size (in bytes)</th>
.
```

```
format STDOUT =
<tr><td> @<<<<<<<<<<<<<<<<<<< </td><td> @<<<<<<
</td></tr>
            $filename                    $filesize
.


$footer = '</table></body></html>';


############################################################
############


print $header;


opendir X, "." or die "Can't open ., $!\n";


for $filename (grep !/^\.\.?$/, readdir X) {
  $filesize = -s $filename;
  write();
}


closedir X;


print $footer;
```

A UNIX Makeover: The Windows Command Line

Warning: This section is included only for fun. If giving your system a lobotomy is not your idea of fun, you may want to skip this section. I've only tested this technique under Windows 98 (and it should be the same under Windows 95). The technique is likely not going to work under Windows NT.

Many people have the misconception that because graphics and the user interface are integrated into the core Windows operating system the shell, Explorer (and its close relative Internet Explorer) is, as well. This section shows how you can create an almost complete replica of the UNIX command line environment in Windows.

Before you attempt this procedure, get a *standalone shutdown utility*. The

normal shutdown utility is built into the Explorer shell, and that's what you're trying to get rid of. You must get a shutdown program so you can log off and on and shut down the system. Obviously, I recommend the one in Essential 97, my own shareware product. I wrote this utility precisely because I like to do things of this nature to the shell. The utility is designed for this kind of work, and it even has an emergency command line.

First, get a UNIX shell. For this experiment, I use a tcsh port to Win32. Tcsh is a UNIX C shell derivative with better support for interactive use. I have used tcsh as my UNIX shell ever since I first encountered it. Tcsh predates the souped-up versions of BASH that are now available. At the time I was introduced to tcsh, the only alternatives were non-standard shells and the enhanced Korn shell. If you prefer BASH or some other shell, use that the shell you prefer. If you don't have a UNIX shell, you can use `COMMAND.COM`.

Next, be sure you have all the UNIX commands. You can get the UNIX commands with the Cygnus port of the Win32 utilities. UNIX shells like tcsh do not have any built-in commands as the Windows command interpreter does. If you want to use commands like `cp` (`copy`) or `mv` (`move`), or even `ls` (`dir`), you need to make the external command available. The only useful command that's built into a UNIX shell is the `cd` command.

Now, edit your `SYSTEM.INI` file. You'll find this file in your Windows directory. (The easiest way to edit it is to click Start/Run… and type `sysedit`.) Find in the `[boot]` section this line:

```
shell=Explorer.exe
```

Replace it with:

```
;shell=Explorer.exe
shell=c:\windows\command\tcsh.exe
```

Adjusting, obviously, the path and filename to point to where your shell is located. (Note that I commented out the existing line instead of deleting it. This strategy is always a good idea.)

Reboot.

When the machine comes back up, you'll get the usual logon prompt, if you have configured your computer for log on. When you log on, you'll see your desktop, but gone will be My Computer and all the icons. Gone will be the Start menu. All you will see is a shell window with a `%` prompt (for tcsh). The main thing to remember is:

DO NOT CLOSE THE SHELL WINDOW!

If you close the shell window, you will be left with a computer that can't do anything. If you inadvertently close it, press Control-Alt-Delete, click Shutdown, and try again.

To run commands, be sure to use the background notation for your shell (putting an `&` after the command in UNIX shells, or using `start` in `COMMAND.COM`), because if you don't, the shell will be useless until you quit whatever program you started. (You have to do the same thing in an `xterm` window on an X Windows system.)

Be certain that some programs (especially setup programs) simply do not run without the Explorer shell. Some will run but will do weird things. Explorer initializes many COM interfaces you probably don't even know exist, and some programs depend on those interfaces. (Word 97 trashed the master document containing this book, for example, when I ran it without the Explorer shell active. I can only guess that the hyperlinks Word creates depend on Explorer's Internet services in some way.) Some inconveniences are inevitable, and I do not mean for this UNIX-like system to be your main working system.

Realize that most programs require the complete, explicit path to themselves in order to run. Very few programs like the Office 97 components will be on your `PATH`. Typing `WinWord` likely won't work, but something like `c:\program files\microsoft office\office\winword.exe` will. Because these long paths are so annoying, if you use this UNIX-like setup for any length of time, you will likely want to create small shell scripts that run common programs for you.

So how do you get out of the UNIX-like system? You need to run the shutdown utility. If you're using tcsh, it will complain that shutdown is not available on Windows 95 (because tcsh thinks you're trying to run the UNIX shutdown command). You must give the explicit path to the shutdown command. In the Essential 97 case, the default directory is your command directory, so the command is:

```
C:/windows/command/shutdown
```

You can also press Control-Alt-Delete and click Shutdown if you get into trouble.

To get out of this mode of operation, edit SYSTEM.INI and change the semi-colon:

```
shell=Explorer.exe
;shell=c:\windows\command\tcsh.exe
```

Remember, changing the shell in this way is meant just for fun. I doubt you will really want to so dramatically alter your working environment, but who knows? This technique may inspire you to bring more UNIX power to your Windows computing experience. You can then have the best of both worlds.

Notes

1. More correctly, from *any* Automation client program. Remember that the next time you need the information or services these objects provide. Using them would be significantly faster in terms of development time.
2. But compare the transition between 16-bit and 32-bit in the PC world: it took about 10 years long than it should have. Inertia is a force to be reckoned with in computing.
3. If you don't know what VMS is, you're lucky: VAX/VMS (Virtual Address eXtension/Virtual Memory System) was a platform/operating system from DEC at the height of the midrange proprietary system craze of the 1980s. (A contemporary was the AS/400 from IBM.) Conspiracy theorists note that each letter in WNT is one greater than VMS, and they draw a comparison with HAL/IBM.

Chapter 11

Developing Perl Programs in Developer Studio

This chapter discusses how to use Developer Studio to develop Perl programs. The material in this chapter originally appeared as the article "Using Perl With Developer Studio 97" in the December 1997 issue of *Windows Developer's Journal*. Since then, I've had more time to experiment with the techniques I introduced in that article and have refined those techniques somewhat. This chapter will take you through the process of adding support for Perl programming to a DevStudio project.

Why Use DevStudio for Perl?

To begin, why would you want to use Developer Studio to develop Perl programs?

One of the main reasons is familiarity. Programmers get attached to their development tools because once you're accustomed to a tool, using it becomes so natural that change is undesirable. UNIX programmers invariably put some form of Emacs or `vi` on their Windows machines and write Perl programs from there. But if you are a native Windows developer who has never used any UNIX tools, such an unfamiliar tool can be intimidating. Even if you *want* to learn a new tool, you may not have the luxury of time to learn it. You could use a third party editor (such as Multi-Edit, which has extensive Perl support), but why not use the same, familiar tool you use for all other development: Developer Studio?

Even if you are interested in writing your Perl programs in some other editor, it's still nice to be able to tweak and run Perl programs from within Developer Studio. Developer Studio has powerful project management facilities for the total project, and it is integrated with source code control systems like Visual Source Safe. Your Perl program may be only one small part of an entire system that involves many modules and programs written in many languages.

Perl may not even be the main point of the project. You may be writing a C++ server program and want to write a test stub to put the server through its paces before a client is even written. Perl is often the perfect tool for writing test stubs. You can whip up a test TCP/IP client or server to exercise a program you are developing, or write an Automation test client, in almost no time.

If you already own a copy of the *Perl Resource Kit* or a separate copy of the

ActiveState Debugger, the good news is you can debug Perl programs in a debugger that looks and feels a lot like the native Visual C++ debugger. If you do not own this commercial product, you can still use the original command line debugger directly from Visual C++.

No, you don't get syntax highlighting. That's the one thing people really want. I do not know how you would extend Developer Studio to highlight a new language. It can obviously be done, since Microsoft did it for Java and HTML. If you could do it, I'm not sure if DevStudio could handle Perl's rather strenuous syntax, which includes regular expressions.

Using a different editor for development doesn't preclude using DevStudio for batch builds and testing. DevStudio can be used to compile anything noninteractively.

Perl Development with DevStudio

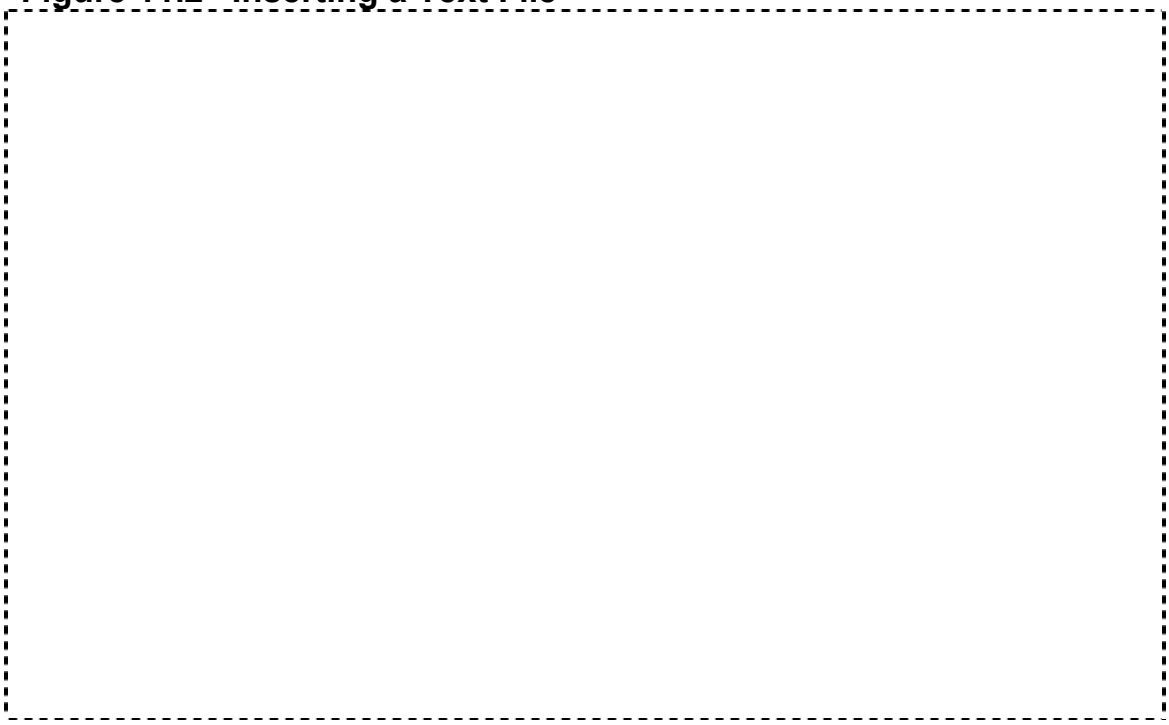
First, I'll develop a Perl program from scratch in DevStudio. This process isn't quite as straightforward as using a Wizard, but it is close, and once you've done it you'll understand the procedure and be able to create new Perl projects very quickly.

Start DevStudio, or close any open workspaces if DevStudio is running. Then go to File/New… and create a new Win32 console mode program, as shown in FIgure 11.1. (If you are using DevStudio 98, be sure to say you want a blank project. Don't let DevStudio 98 generate a C++ skeleton application. DevStudio 97 doesn't have this extra step.)

Figure 11.1 Creating a Project in Developer Studio

Now that you have a blank project, Go to File/New… again and insert a text file (see Figure 11.2). Name it with a `.pl` extension. (DevStudio doesn't care if you create a text file that does not use the `.txt` extension.)

Figure 11.2 Inserting a Text File



Syntax Checking a Perl Program

At this point, you have a plain text file you can edit and save but not execute. The next step is to add syntax checking for the Perl source file. In the Workspace window, go to the FileView tab, select the `.pl` file, right click, and choose Settings (see Figure 11.3).

Figure 11.3 Adding Syntax Checking

Tab over until you get to the Custom Build Step. Under it, fill out these fields:

- Description: Perl Syntax Check on `$(InputPath)`
- Commands: `c:\Perl\bin\perl -c $(InputPath)`
- Outputs: `tmp.out`

You will, of course, have to adjust the path to where you installed the Perl interpreter.

Running a Perl Program within DevStudio

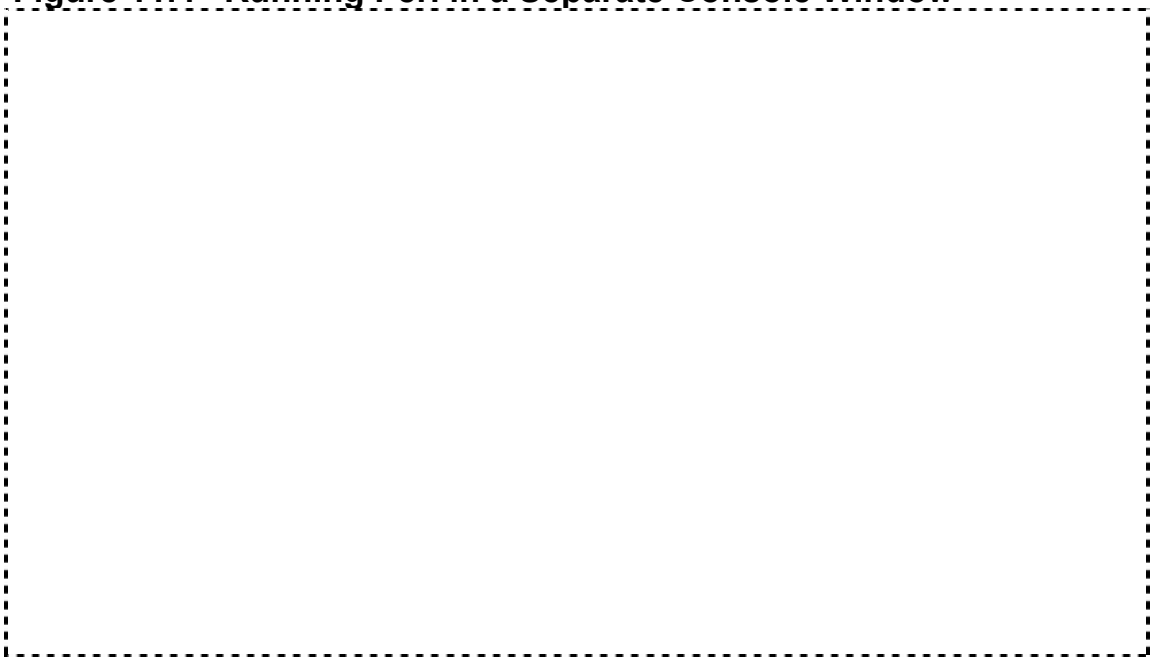
If your program has just a little output, you can run it and have its output appear in the Build window.

Simply delete the `-c` from the Commands part of the Custom Build Step dialog. The Build window will preserve about 8000 lines of output before truncating old lines to make room for new lines. For programs that have little output, this feature is convenient. For other programs, too much output is produced for it to be practical. For a program like a TCP/IP server, it won't work at all.

Running a Perl Program outside DevStudio

Remember when you created the project, you created a project for a Win32 console mode program. That's what the Perl interpreter is. All you have to do to run the Perl program under the interpreter is make DevStudio think that the Perl interpreter is the project's end result. To run the Perl program in a separate console window using DevStudio's Build/Execute command, simply change the executable that the project thinks its end result is to `perl.exe` (see Figure 11.4). Since you have no source files to rebuild to produce the executable, Developer Studio just runs the executable.

Figure 11.4 Running Perl in a Separate Console Window



Go to Project/Settings… and tab over to the Debug tab. Under it, fill out these fields:

- Executable for debug session: `c:\Perl\bin\perl`
- Program arguments: `test.pl`

The executable is `perl.exe`, and the argument is the script.

Adding the ActiveState Debugger

You can also add a debugger to the DevStudio configuration. For this section, I assume you have either a standalone copy of the ActiveState Debugger or a copy of the *Perl Resource Kit* for Win32, which includes the debugger. If you do not, `perl -d` will still *work* with DevStudio. (`perl -d` will just pop open the original Perl command line debugger). For GUI debugging such as the kind performed in DevStudio, ActiveState's debugger is the only option as I write this book.

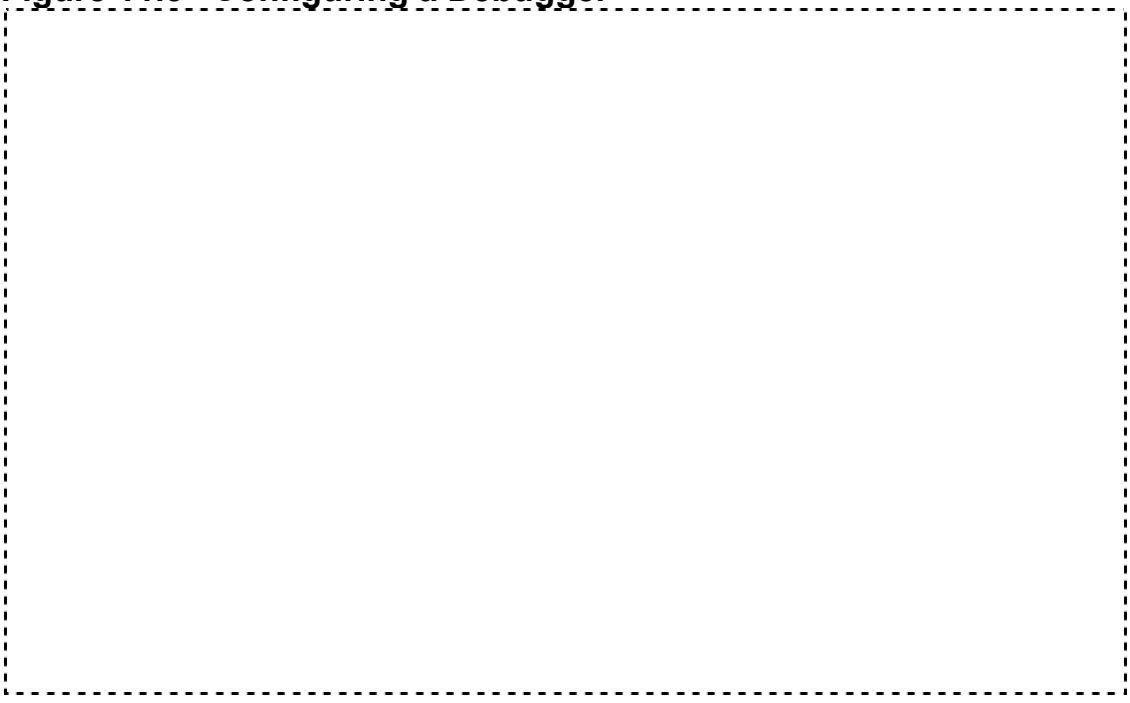
To install the debugger under the Tools menu, go to Tools/Customize and tab over to the Tools tab (see Figure 11.5). At the bottom of the list of tools, add a new entry named ActiveState Debugger.

Fill out these fields:

- Command: `c:\Perl\bin\perl.exe`
- Arguments: `-d $(FileName)$(FileExt)`

When you choose Tools/ActiveState Debugger, the debugger will run and debug your source file.

Figure 11.5 Configuring a Debugger



Bibliography

Like hammers and chisels to wood carvers or wetsuits and oxygen tanks to deep-sea divers, books are the essential tools for the computer programmer. When crafting programs, the right books can make all the difference in the world. Good reference material means productivity. I have not only listed the books useful to me in preparing this book but have added my personal annotations.[1]

Abrahams, Paul W., and Larson, Bruce A. 1996. *UNIX for the Impatient*, Second Edition. Reading, Massachusetts: Addison-Wesley. (My college networking and system administration professor, Dr. J. Dean Brock of the University of North Carolina at Asheville, always recommended this book highly to students faced with UNIX for the first time, and in the years since I have yet to find any better treatment of UNIX for the end user and neophyte. If you need background information on the UNIX shell, the vi editor, the UNIX utilities, etc., this is a good starting point.)

Automation Programmer's Reference. 1997. Redmond, Washington: Microsoft Press. (The foundational work on Automation published by Microsoft. Although it concentrates on servers rather than clients the book is interesting because it explains the background underlying such fundamental Automation concepts as IDispatch and dual interfaces which high-level development tools hide. Good reading for the programmer who wants to know how things really work. This book is extremely technical and not for the timid.)

Bocktor, David. 1997. *Microsoft Office 97 Visual Basic Step by Step*. Redmond, Washington: Microsoft Press. (Excellent tutorial introduction to Office 97 programming, a good companion for the *Microsoft Office 97/Visual Basic Programmer's Guide* or the *Unleashed* book. For Perl programmers, it has a whirlwind introduction to VBA coding. Anyone who has ever used BASIC and knows OOP theory should be able to pick up VBA quickly.)

Box, Don. 1998. *Essential COM*. Reading, Massachusetts: Addison-Wesley. (The single most essential book on COM for the Windows programmer to own. This book contains the best explanation of the COM philosophy I've ever read. If you want to *understand* COM rather than merely use it, you need this book.)

Brockschmidt, Kraig. 1994. *Inside OLE 2*. Redmond, Washington: Microsoft Press. (*Inside OLE 2* is an older book, but COM has not fundamentally changed since the book's introduction. This book has, until recently, been the only book on COM, which could explain COM's slow acceptance among developers since this book is generally considered unapproachable on account of its complexity.)

Cant , Marco. 1997. *Mastering Delphi 3*, Second Edition. Alameda, California: Sybex. (Good discussion of Delphi's Automation support. Also good as a general Delphi reference.)

Christian, Kaare. 1992. *The Microsoft Guide To C++ Programming*. Redmond, Washington: Microsoft Press. (Original book on C++ from Microsoft which went along with their initial C++ implementation in the classic Microsoft C/C++ 7.0. Discusses MFC without CCmdTarget, and it is interesting to compare the class diagrams to today's UML.)

Cornell, Gary. 1998. *Learn Microsoft Visual Basic Scripting Edition Now*. Redmond, Washington: Microsoft Press. (I like this book because it is a fast, painless, and friendly crash course in VBScript. Its drawback is that it assumes VBScript will be used for web client programming, and the entire book is written from that perspective.)

Cowart, Robert. 1998. *Mastering Windows 98*. Alameda, California: Sybex. (This mammoth book, which is almost a *cube*, contains a good treatment of the Windows Scripting Host, one of the few such I have been able to find. Its comprehensiveness has made it the only book I've needed for Windows 98 use and administration.)

December, John, et al. 1996. *HTML 3.2 & CGI Unleashed*, Professional Reference Edition. Indianapolis, Indiana: Sams Publishing. (I've read a *lot* of web books, and this is the best one. Even though it is somewhat outdated, it is still my favorite because it is the first systematic treatment of web programming as a discipline in the field of computer science, that I've seen. It comprehensively covers web design and implementation in all phases from client to server. Plus, it's not like CGI has changed all that much! If you ever see this book, be sure to get a copy.)

De la Cruz, Irving, and Thaler, Les. 1996. *Inside MAPI*. Redmond, Washington: Microsoft Press. (The foundational book from Microsoft on MAPI. Basically, reading this will cure you of ever wanting to use MAPI. Not for the timid.)

Dougherty, Dale, and Robbins, Arnold. 1997. *sed and awk*, Second Edition. Sebastapol, California: O'Reilly and Associates. (A look at two early UNIX utilities which had profound influence on Perl. Interesting to contrast Perl with these – the differences are instructive. While sed is primitive and difficult to use, awk is a significant language that was a direct influence on Perl. The awk language is one of the first high-level text processing languages of non-mainframe origin.)

Eddon, Guy, and Eddon, Henry. 1998. *Inside Distributed COM*. Redmond, Washington: Microsoft Press. (This book replaced *Inside COM* as Microsoft's official COM tutorial book. It is newer and covers more of the late additions to support

distributed computing. As *Inside COM* becomes harder to find, I find myself recommending this book to supplement *Essential COM*.)

Friedl, Jeffrey E. F. 1997. *Mastering Regular Expressions*. Sebastapol, California: O'Reilly and Associates. (This is the definitive book on the subject, and one that any Perl programmer who wants to understand regular expressions needs to read. Although the book covers regular expressions in general, it uses Perl as its main example of a regular expression parsing engine.)

Goetter, Ben. 1996. *Developing Applications for Microsoft Exchange with C++*. Redmond, Washington: Microsoft Press. (The only explicit mention of `MAPI.Session` in any Microsoft literature I've ever seen is in this little-known book, p. 79-80. By the way, if *Inside MAPI* doesn't cure you of wanting to use MAPI, this book will. It is ten times as technical. *But*, the first few chapters have a lucid overview of MAPI. It gets progressively more technical as you continue.)

Grimes, Richard, et al. 1998. *Beginning ATL COM Programming*. Chicago, Illinois: Wrox Press. (At the time of this writing, the best and just about the *only* treatment of ATL available.[2] I used this book in writing the PerlX C++ client. Note that this book discusses client programming in C++. If you get a book, be sure it treats both client and server programming. Most C++ books tend to discuss only the server end.)

Jamsa, Kris. *1994. Concise Guide to MS-DOS Batch Files*. Redmond, Washington: Microsoft Press. (Although old, this little book is one of the best batch file books ever written. Even today, sometimes tasks are most easily and naturally expressed as batch files despite having other languages available, particularly for true *batch* jobs where programs need to be run in sequential order.)

Kespret, Istok. 1996. *Zip Bible*. Grand Rapids, Michigan: Abacus. (The best ZIP reference I've found. The only problem is it assumes use of PKZIP instead of InfoZip. This book was also published originally as *PKZIP, LHARC, and Co.* and some copies may still be circulating with that name.)

Kruglinski, David J., et al. 1998. *Programming Microsoft Visual C++*, Fifth Edition. Redmond, Washington: Microsoft Press. (This is the single most essential book for Visual C++ programmers to have. It covers Automation DLLs and just about everything else. Note that the edition numbers do not synchronize with the Visual C++ product release numbers: the fifth edition of this book covers the Visual C++ 6.0 release.)

Leber, Jody. 1998. *Windows NT Backup and Restore*. Sebastapol, California: O'Reilly and Associates. (This book picks up where I leave off and discusses large-scale backups.)

Lomax, Paul. 1997. *Learning VBScript*. Sebastapol, California: O'Reilly and Associates. (Best all-around introduction to VBScript. Good because it does not assume VBScript will be used for web client programming. Most VBScript today is not web client programming.)

McConnel, Steve. 1993. *Code Complete*. Redmond, Washington: Microsoft Press. (Although I do not use this book directly in the text, it has certainly impacted my programming. I wish there were some way to make every programmer read this book before writing software! If you only read one book on programming in your life, be sure it is this one.)

McConnel, Steve. 1996. *Rapid Development*. Redmond, Washington: Microsoft Press. (The thesis of this book is that, just as better algorithms improve performance more dramatically than code bumming, better development methodologies improve project schedules more than silver-bullet tools. This book is essential reading because it has a treatment of smoke testing, and many other subjects.)

McConnel, Steve. 1998. *Software Project Survival Guide*. Redmond, Washington: Microsoft Press. (McConnel changed his modus operandi somewhat by making this book shorter and less exhaustively documented by references. It sums up much of his previous two books in the context of a software project. Smoke testing, as well as many other subjects, appears.)

McFreides, Paul. 1997. *Visual Basic for Applications Unleashed*. Indianapolis, Indiana: Sams Publishing. (This impressively comprehensive book, by a single author, has proven to be an invaluable resource for my Office 97 efforts. Its comprehensiveness makes it a good one-volume reference library to use while programming.)

Microsoft Office 97/Visual Basic Programmer's Guide. 1997. Redmond, Washington: Microsoft Press. (The definitive Office 97 programming book. Of interest to Perl programmers are complete maps of the object models for various Office applications. The bad news is the properties and methods for these objects are not listed.)

Microsoft Press Computer Dictionary, Third Edition. 1997. Redmond, Washington: Microsoft Press. (Well done and amazingly neutral dictionary, considering the source. Good contrast to the Jargon File.)

Mitchell, Stan. 1997. *Inside The Windows 95 File System*. Sebastapol, California: O'Reilly and Associates. (An interesting discussion of the installable file system architecture used by such tools as ZipMagic 98. Even if you're not going to write a file system, it's interesting to see how the architecture works. Not for the timid, this is a very technical book.)

Orfali, Robert, et al. 1997. *Instant CORBA*. New York, New York: John Wiley & Sons. (Good tutorial introduction to CORBA, which is similar to COM but more complex and with less mature tools. It is interesting to compare and contrast COM with CORBA, and this book is the most approachable discussion of CORBA I've found.)

Perl Resource Kit, Win32 Edition. 1998. Sebastapol, California: O'Reilly and Associates. (If you're serious about Perl programming, this kit is important to have since it includes good documentation on the Win32 Perl. It also has such nice things as the graphical Perl debugger.)

Peschko, Edward S., and DeWolfe, Michele. 1998. *Perl 5 Complete*. Berkeley, California: Osborne/McGraw-Hill. (Contains the best Win32 coverage of any Perl book I have seen, and is an okay reference.)

Plauger, P. J. 1992. *The Standard C Library*. Englewood Cliffs, New Jersey: Prentice Hall. (Good discussion of the C time library, the basis for Perl and many other languages' time functions.)

Powell, Thomas A. 1998. *HTML: The Complete Reference*. Berkeley, California: Osborne/McGraw-Hill. (As I realized my favorite web book, *HTML 3.2 & CGI Unleashed*, was becoming obsolete in some areas, I supplemented it with this excellent book. Of the 4.0 generation books about HTML, this is my favorite. As this book was going to press, a second edition of *HTML* was released, which is even better.)

Rahmel, Dan. 1998. *Visual Basic Programmer's Reference*. Berkeley, California: Osborne/McGraw-Hill. (Indispensable reference book similar to the *Perl 5 Pocket Reference*. Occasional VBA and VBScript users especially will benefit from having a copy to look things up.)

Richter, Jeffrey. 1998. *Advanced Windows*, Third Edition. Redmond, Washington: Microsoft Press. (Richter's book is to Win32 what Stevens' book is to UNIX, the definitive systems programming book for the platform. For Perl programmers who are migrating from one platform to another, or use both, an in-depth understanding of issues such as the process model is important.)

Rogerson, Dale. 1997. *Inside COM*. Redmond, Washington: Microsoft Press. (A good COM overview similar to *Essential COM*'s in the material it covers. I suggest reading both for the different perspectives they give.)

Schwartz, Randal, et al. 1997. *Learning Perl on Win32 Systems*. Sebastapol, California: O'Reilly and Associates. (The best general introduction for Perl beginners I have seen aimed at Windows users. If you need a first book on Perl, this is it.)

Solomon, Christine. 1997. *Microsoft Office 97 Developer's Handbook*. Redmond, Washington: Microsoft Press. (Solomon advocates looking at Office 97 from a code-reuse standpoint and creating business solutions by customizing Office rather than reinventing the wheel. There are also Excel 97 and Access 97 *Developer's Handbook*s, and this one is harder to find than those for some reason. Generally, I get the impression *Microsoft Office 97 Developer's Handbook* is an overlooked book, and deserves more recognition than it has gotten for its code-reuse perspective. If you want information on using Office from a code-reuse standpoint, this book is worth reading.)

Stevens, W. Richard. 1992. *Advanced Programming In The UNIX Environment*. Reading, Massachusetts: Addison-Wesley. (Stevens' book is to UNIX what Richter's book is to Win32, the definitive systems programming book for the platform. For Perl programmers who are migrating from one platform to another, or use both, an in-depth understanding of issues such as the process model is important.)

Tornsdorf, Manfred, and Tornsdorf, Helmut. 1991. *Batch File Power Tools*. Grand Rapids, Michigan: Abacus. (This beloved and ancient book in my library inspired TCD95 with its TCD command. The TCD program came on the companion 5 1/4-inch disk. This book, full of utilities and tools, reminds me of what's missing from today's sterile, homogeneous Windows experience. I hope my book rekindles some of the exploring and hacking just for the fun of it that went on in earlier times.)

Tornsdorf, Manfred, and Tornsdorf, Helmut. 1992. *DOS 5.0 Complete*. Grand Rapids, Michigan: Abacus. (This book also came with the TCD command on the disk and would likely be easier to find than *Batch File Power Tools*. There is also a DOS 6.0 edition that I do not have. In addition to having the TCD command, the book is a good DOS command reference still useful for Windows 9x and batch file programming.)

Vromana, Johan. 1998. *Perl 5 Pocket Reference,* Second Edition. Sebastapol, California: O'Reilly and Associates. (So essential that I own *two* copies of this: one for my briefcase, and one by my computer. 'Nuff said.)

Wahli, Ueli, et al. 1997. *Object REXX for Windows 95/NT*. Upper Saddle River, New Jersey: Prentice Hall. (Definitive Object REXX book for Win32 platforms. The book comes with the Object REXX interpreter on the CD-ROM, which makes it ideal if you want to explore REXX. Surprisingly, the REXX icon editor is a repackaged version of the old Borland Resource Workshop.)

Wall, Larry, et al. 1996. *Programming Perl*. Sebastapol, California: O'Reilly and Associates. (A.k.a. "the camel book", this is *the* fundamental Perl book. It is an excellent and authoritative reference for Perl programmers. Tutorial information is generally regarded as being a bit advanced in its presentation, so it may not be the best book for a total neophyte, particularly one who has no previous UNIX experience. Once you know enough Perl to hold your own, though, this is the one essential book.)

Zaratian, Beck. 1997. *Microsoft Visual C++ Owner's Manual*. Redmond, Washington: Microsoft Press. (This is the *only* book I have ever seen which discusses the Visual C++ Automation interface. *Note*: This book has been republished as part of the Visual Studio 98 documentation set. From what I can tell, the book changed very little from this edition.)

Notes

1. If this list isn't enough, see my online book review site *The Cyber Reviews* at http://cyberreviews.skwc.com/, where I hold forth at longer length and in more detail about these books and any others I can get my hands on.
2. I find it baffling that Microsoft has introduced ATL, a new technology it wants people to use, and has not published (at the time of this writing) a book on it—despite having its own publishing company. I'm not sure why Microsoft doesn't want to evangelize ATL, but this pattern of introducing a new technology without any real educational resources is a recurring one. COM and ATL were introduced with little fanfare and with little information for people who wanted to know more.

About The Author

Scott McMahan has been writing as long as he has been working with computers, and both began in junior high school back in 1984. The writing was then fiction, and the computer was a TRS-80 CoCo 2 from Radio Shack. He began using Perl around 1992 when he took over The Genesis Mailing List (the only remaining vestige of this list today is *The Genesis Discography*, `http://cyberreviews.skwc.com/genesis.html`, which Scott maintains to this day) and found some old Perl code for automating a mailing list. The code only did so much, and Scott had to either spend his time manually running the list or learn Perl and upgrade the software. Learning Perl sounded more interesting, so he did. Scott turned this project into his senior research at the University of North Carolina at Asheville. No one there knew Perl nor mailing lists very well back in those days, but his research sounded good enough for them to give him a BS degree in computer science. From there he went into the world of work, taking Perl with him, and using it every chance he got. He wound up at SoftBase Systems, where he is now employed, as both a webmaster/network administrator and programmer. Wearing many hats in a small company led Scott to automate everything he could, which gave rise to most of the material in this book. Scott's most recent Perl exploit has been embedding Perl into SoftBase Systems' NetLert Enterprise Server (about which you can learn at `www.netlert.com`) as its scripting language.

Besides being a computer programmer, Scott is an avid reader (ranging from computer programming books to classical literature and everything in between) and writer (his web site, `cyberreviews.skwc.com`, puts the "vanity" back into vanity publishing with his essays, stories, and poems). He is a sports fan, enjoying the Carolina Panthers, Atlanta Braves (for which he maintains a page on the whereabouts of former players, see `cyberreviews.skwc.com/wherebrave.html`), Atlanta Hawks, and UNC Tarheels. Much of his writing is done while watching televised games out of the corner of one eye. Scott's all-consuming, Gaudi-sized magnum opus will be an autobiographical web site about his life and experiences, which was supposed to be done a year ago but may never be finished.

Index

Symbols

Numerics

A

B

C

D

E

F

G

H

I

Q

R

S

T

U

V

W

X

Y

Z

What's on the CD-ROM?

Automating Windows with Perl includes a companion CD-ROM with all the Perl programs and other programs developed in the book.

The Perl programs, for the most part, are so short that there has been no attempt to make them run under `strict` and `-w`. The rest of the programs are either short demonstrations of specific ideas or building blocks that are the starting points towards bigger and more complete programs, which would then be made to run under `strict` and `-w`.

The code has been tested with the following development tools:

- All the Perl programs were written and tested under Perl v5.005 (using ActiveState Build 504). They should work on all higher versions, but they will not work under lower ones.
- The C++ code was all compiled under Visual C++ v6.0. The code should work under Visual C++ v5.0 as well, but it has not been tested under v5.0. The C++ code probably will not work under pre-v5.0 Visual C++.
- The Delphi code (what little bit of it there is in the book) has been tested under Delphi 3.
- The Visual Basic for Applications code was all developed and tested under the various Office 97 applications. It will not work under any previous versions of Microsoft Office, but it will probably work under later versions.

Also included on the CD is a bonus program from the author's Essential 98 suite, which will be useful with some of the Perl programs developed in this book. Everyone who buys this book gets a free, registered copy of Run Control 98. This program is the missing piece of Windows 9x (it works under both 95 and 98) that allows you to control which programs are run automatically when the machine boots.

For additional information on the software, projects, and source code for ***Automating Windows with Perl***, see the `readme.html` file on the CD-ROM.