微软编程圣典丛书（影印版）

Microsoft

配光盘

Microsoft Windows 国际化程序设计

（影印版）

Put Windows 2000 to work

Features information to help you develop and deploy applications for windows 2000

International Programming for Microsoft Windows

- 来自微软的第一手技术资料，深入专业的编程技术
- 丰富的 VC++ 6.0程序实例
- 软件国际化和本地化的权威指南
- 高级程序员必备

[美] David A. Schmitt 著

北京大学出版社

http://cbs.pku.edu.cn

微软编程圣典丛书(影印版)

Microsoft Windows 国际化程序设计

Microsoft 公司 著

北京大学出版社

内 容 简 介

本书是《微软编程圣典丛书（影印版）》之一，讲述如何开发易于本地化的国际化应用程序，内容涉及VC++的使用、MBCS、Win32 NLS API、多语言API等。为了增加本书的实用性，特以配套光盘的形式提供了丰富的程序实例以及本书的电子版。

本书由微软公司组织专家编写，具有相当的技术深度，是中、高级程序员必备的参考书。

书　　名：Microsoft Windows 国际化程序设计（影印版）
责任著作者：Microsoft 公司 著
标 准 书 号：ISBN 7-900629-39-4/TP・33
出 版 者：北京大学出版社
地　　址：北京市海淀区中关村北京大学校内　100871
网　　址：http://cbs.pku.edu.cn
电　　话：出版部 62752015　发行部 62765127　62754140　编辑室 62765127
电 子 邮 箱：wdzh@mail.263.net.cn
印 刷 者：北京大学印刷厂印刷
发 行 者：北京大学出版社
经 销 者：新华书店
787 毫米×1092 毫米　16 开本　29.875 印张　1200 千字
2000 年 9 月第 1 版　2000 年 9 月第 1 次印刷
定　　价：88.00 元

丛 书 序

世纪交替，IT 产业更加步履匆匆。

Microsoft 公司早已以其在编程方面的非凡成就闻名于世，并树立了在计算机软件领域和发展史上不可动摇的地位。毋庸置疑，该公司技术上的优势是其获得成功的重要因素之一。今天，它的技术不但已经变得非常强大，而且具有惊人的发展速度。尤其是 Windows 2000 技术的推出，更是展示了 Microsoft 的无穷魅力，它突然间提供了如此丰富的新特性，使我们仿佛在一瞬间便被淹没在 Windows 2000 浩瀚的技术海洋之中！

工欲善其事，必先利其器。作为 Windows 应用程序设计人员，必须紧密跟踪 Microsoft 公司的最新技术，深入 Microsoft Windows 编程的内幕，掌握关键的编程技术。这套《微软编程圣典丛书（影印版）》的推出，就是为了向有关的专业人员全面推介微软编程的核心技术，以便于他们设计高质量的 Windows 应用程序。

Microsoft 技术博大而精深，绝非某个人在短时间内所能掌握。为此，特按照技术上的逻辑关系组织成 9 个相对独立的部分，分别涉及基于服务器的应用程序、COM+基本服务、Windows 网络编程、国际化程序、MFC、Windows 编程、服务器端应用程序、Outlook 与 Exchange 编程、驱动程序模型等。每一部分的内容独立成册，集中讲述一组相关的编程技术。这套《微软编程圣典丛书（影印版）》共 9 本。特定编程领域的专业人员可以从中选取自己需要的一本或几本，使学习过程更加快速、省时、有效而直观。

这套丛书中的任何一本都涉及一门完整的编程技术，因此有着相当的深度，而且内容比较丰富。为了避免将其写成深奥而抽象的理论书，特在书中适当的位置穿插进许多贴切的程序实例。另外，每本书都有配套的 CD-ROM，内有书中的程序实例和本书的电子版。

本套丛书由 Microsoft 公司组织相关领域的专家编写。他们深谙 Microsoft 的编程技术内幕，具有丰富的程序开发经验，所以，这套丛书是他们智慧的结晶，是该领域极具权威性的著作，堪称独领风骚。

鉴于此，特向中、高级 Windows 应用程序设计人员郑重推荐这套佳作！

出版者

2000 年 9 月

Preface

At the start of my programming career more than twenty-five years ago, most American software designers were blissfully ignorant of computer usage outside the United States. There was almost no market for packaged software products because the typical program was developed in house by the company using it. Although the largest firms were multinational, their various, far-flung divisions usually took care of their own software development, often by tediously adapting software originally written by Americans for Americans.

Over the years, I gradually became aware that programmers outside the United States were encountering serious problems as they tried to adapt American-designed software for use in their own countries. For example, at a UNIX conference in Copenhagen I heard European programmers bitterly complain about the American biases built into that operating system. Among their complaints was the pervasive use of non-European characters throughout UNIX and the C language. I learned that in some countries, important C programming characters such as the pound sign (#) and square brackets ([ ]) don't even appear on the most popular keyboards!

Complaints about what I call "chauvinistic programming" swelled during the 1980s as personal computers appeared on desktops throughout the world and as a vigorous software product industry emerged. I've heard dozens of stories about how a clever programmer cobbled up a kludge in order to get some essential software product working in his or her local environment.

For instance, back in 1984 a friend in Stockholm showed me a little MS-DOS TSR (terminate and stay resident) program he had written to make Lotus 1-2-3 properly display the various accented Scandinavian characters. Lotus was so busy with its exploding American market that they couldn't take the time to change 1-2-3 internally for European character support. Accented characters such as Ä could be entered from the Swedish keyboard, but they showed up on the screen as gobbledygook. My friend solved the problem by having his resident program periodically scan the video memory and convert the special characters to the correct form. Although the solution worked, it was somewhat disturbing to see solid garbage on the screen dissolving into recognizable Swedish as you typed.

Whenever a non-American related one of these stories, the point was usually quite simple: *Programmers should consider international applications in the fundamental software design stage, not as an afterthought.*

This is particularly important now that personal computers are being used by ordinary people who have neither the time nor the background to deal with esoteric computerese. These users want the computer to be a simple tool that is easy to understand and operate. They aren't getting paid to read large manuals and figure ways to overcome the limitations imposed by inadequate hardware and software. They demand that their interface with the computer be conducted in their native language. Ultimately, they will buy and use software only if it feels natural to them.

The more I understood this situation, the more confused I became. After all, like most U.S. programmers, I speak only English. Sure, my four years of high school Latin enable me to figure out the origin of many English words, to find my way around the Milan subway system, and even to make some sense out of Italian or Spanish newspaper headlines. But it appeared that I would have to become fluent in several languages in order to design international software. And while I might be able to learn one or two European languages, others such as Japanese, Chinese, and Arabic looked like formidable barriers.

These depressing thoughts led me to undertake the research which resulted in this book. The first glimmer of hope came when my company's Japanese agent assured me that American programmers don't need to understand Japanese in order to write software that can cope with that country's language and customs. Checking around with other friends and acquaintances throughout the world, I found that they generally agreed: *By following the proper design and coding rules, any programmer can write multilingual software.*

This was really good news! But what are those rules? To discover them, I poked around in the available literature and attended an excellent seminar on Asian dual-byte character sets sponsored by IBM's World Trade division. Through these activities, I discovered that much work has been done in this area. However, the results are scattered throughout documents such as the ANSI C and C++ standards and manuals produced by IBM, Microsoft, and other computer and operating system vendors.

This book is my attempt to pull these diverse sources together and present a methodical approach to international programming for the Microsoft Visual C++ programmer. I decided to concentrate on Visual C++ for several reasons. First, it's the development environment I use and teach most frequently. Second, it's the most popular tool for producing industrial strength Microsoft Windows applications. And finally, I believe that Windows, C++, MFC, and ATL represent the state of the art in international programming.

Acknowledgments

Every technical book is a team effort, but none more so than one which covers a broad topic such as international programming. I was fortunate to work with an excellent team at Microsoft Press, beginning with Ben Ryan, who encouraged me to tackle this daunting topic, and Kathleen Atkins, who applied gentle pressure and positive feedback throughout the 18 months it took me to get the job done.

Julie Xiao went beyond the call of duty. She dug out the "true facts" to correct my technical errors and misconceptions, fleshed out topics where my treatment was too light, and verified each of the sample programs. Naturally, these programs were bug free when I delivered them, but Julie still found some errors. Many thanks, Julie.

Michelle Goodman, Jennifer Harris, and Kathleen Atkins (wearing another hat) ensured that the words formed sentences, that the sentences formed paragraphs, and that they all made sense. Patricia Masserman proofread them all. As with the sample programs, my prose was "perfect" when I delivered it, but somehow the team made it better. Many thanks to you, also.

Of the many people who gave this book its look and feel, Gina Cassill, Michael Kloepfer, and Joel Panchot are most responsible for its interior. If you're standing in a bookstore right now and you grabbed this one off the shelf (because of its eye-catching cover), and you like the figures and overall layout, you're appreciating their work. I know I do.

I also want to thank the people outside Microsoft Press who gave their advice and assistance. Carter Shanklin's courtesy enabled me to get the project started. Avery Bishop and Lori Brownell of Microsoft's Globalization Team willingly answered my questions about Windows 2000, Uniscribe, and other international programming topics. P. J. Plauger, Angelika Langer, and David Smallberg shared their deep knowledge of standard C and C++.

Special thanks to many of my students who encouraged me and contributed interesting anecdotes and useful advice from their diverse ethnic backgrounds. One of the nicest aspects of teaching COM and other subjects for DevelopMentor is that I get to meet so many interesting people.

Finally, after all the words I wrote in this book and others, I've found none to express my gratitude and love for my wife, Karen. I only hope that actions say what words cannot.

David A. Schmitt
January, 2000
Saint Louis, Missouri

Introduction

Nicolas Chauvin was a French soldier known for his excessive devotion and loyalty to Napoleon during the early years of the nineteenth century. His name is the root of the English word *chauvinism,* meaning blind patriotism or partiality. Unfortunately, we've recently warped this fine word so that many believe it characterizes only certain male attitudes toward females. Chauvinism, however, is the perfect way to describe software development practices that make it difficult to adapt programs for use in other countries.

Most of us must plead guilty to some level of chauvinistic software design techniques. We embed prompters and other message text deep in our code so that translation becomes a major programming effort. Our screen and report layouts don't allow translators to rearrange fields or change their lengths to accommodate other languages and cultures. We often use algorithms for sorting, scanning, and generating text that rely upon a specific character set (usually ASCII or EBCDIC) or a specific format, and these algorithms fail when a "foreign character set" or "foreign formats" are introduced.

Software product companies now recognize the need for more cosmopolitan design techniques because of the potential for sales in other countries to people who can't or won't stray very far from their native tongue. Also, multinational corporations are realizing that they can minimize programming costs by developing common software for their divisions and subsidiaries. In both situations, we're encountering a new type of user quite different from the highly trained and specialized personnel who traditionally have worked with computers.

For many years, English was a kind of universal language, a *lingua franca* or an *Esperanto,* among computer programmers, operators, and users. Most people who had day-to-day contact with computers weren't too upset when error messages, manuals, and control panel markings weren't in their native language. They usually had enough education to cope with English, and many of them were fluent in it. If they didn't have that skill, they gradually became proficient as they wrestled with English-oriented computer manuals and programming languages. They were technical types, like us.

The personal computer has caused this situation to change dramatically. Today's typical PC users want to work mainly in their own languages. For typical PC users, the PC isn't an object of veneration as it has been for us technical types. Rather, it's merely a piece of office equipment like a copier or a calculator. They see it as a tool, a means to an end, and they won't waste time with difficult tools. They demand that computers have standard human interfaces like the ones they've come to expect in cars, telephones, and other everyday instruments. They don't want to read a lengthy manual in order to use a program, and they don't want to communicate with the computer in a foreign language.

I've often heard programmers express disdain for this new breed of non-technical computer users. "After all," we geeks say, "the computer isn't a toy; it's a sophisticated piece of equipment, and you have to pay your dues in order to understand it. These users should at least learn how to resolve IRQ problems!" But how many of us understand the inner workings of our cars, televisions, microwave ovens, and camcorders? Do we have much patience when it comes to reading the owner's manual? Would we be happy if all of the instructions and markings were printed in Japanese?

So any software developer who continues to cater only to the technical computer user is ignoring the mainstream of our business. Clearly, the needs of the new user community require that programmers learn more cosmopolitan software design techniques. Even if the resulting programs are larger, slower, and more expensive to develop, modern computer applications demand international solutions, and rapid hardware evolution will compensate for the cost of internationalization.

This book shows how you can become a proficient international programmer by using design techniques that will enable your programs for use in other countries. You'll find that these techniques aren't much of a burden because modern operating systems and programming languages do a lot of the work for you.

Chapters 1 and 2 describe the issues you'll encounter when designing international software. This section of the book takes a broad look at the major language and cultural groups to show how their differences impinge upon your programming activities.

Chapter 3 then traces the history of character sets to show how this evolution has influenced international communication in general and computer programming in particular. Chapter 4 explains how these character set issues affect standard C and C++ programs.

Chapters 5 and 6 explore the standard "locale" feature for C and C++, and Chapter 7 describes the related Microsoft Visual C++ extensions that can simplify international programming.

Chapters 9 and 10 examine the rich set of National Language Support (NLS) features that Microsoft has provided in the Win32 operating system environment. These features go well beyond standard C and C++ by supporting advanced multilingual applications with a graphical user interface. Because the Win32 application programming interface (API) uses a C language interface, C and C++ programmers can call Win32 functions directly.

Chapter 11 concludes by presenting a set of guidelines for developing international software. With these in mind, you should be able to make the proper trade-offs between program complexity and geographic scope.

Table of Contents

Chapter 1

The Basic Issues

The challenge before us is to design software that can be easily adapted for use in other countries. To face this challenge, we programmers must first gain a general understanding of the cultural differences that have an effect on software design. This chapter and the next two describe the most important of these differences.

I must make a disclaimer at this point. Although I've traveled outside the United States many times (primarily in Europe and Japan), I'm neither a linguist nor a cultural expert. Sure, I can decipher European road maps[1] and find my way around the Tokyo subway system. I've also become pretty adept at handling foreign currency and don't get cheated too much when haggling with a street vendor. I've never missed a meal (although I probably should) because there's usually something on the menu that I understand. (Besides, every place I've visited has had a McDonald's nearby.)

In other words, my knowledge of non-U.S. locales is about the same as any other well-traveled U.S. businessperson. So how can I presume to write this book? Well, I've seen many examples of both good and bad international software, and I firmly believe that you don't need to be multicultural, multilingual, or a "Renaissance dude" to enable your software for global usage. You must understand the "rules of internationalization," however, and the technology that supports them. Then by diligently following the rules, you will produce generalized programs that other software experts can adapt to specific languages and cultures.

---

1. I must thank the good brothers at Saint Mary's High School who forced me to study Latin for four years. Since many European languages are based on this "dead language," I've found it amazingly easy to decipher maps, road signs, and newspaper headlines. Of course, occasionally I get the wrong meaning, such as that time I drove out of the Frankfurt Airport parking garage via the entrance ramp.

LOCALE-DEPENDENT SOFTWARE

At the risk of stating the obvious, I'll begin by observing that any program with a nontrivial human interface is likely to be locale dependent. Humans usually communicate through words, which are written or spoken, and pictures, which are either drawn or portrayed by some form of body language. This interface works best if it's instinctively familiar to both parties in the conversation—that is, if both people share the same language and culture. For example, the offensive hand gesture known as "flipping the bird" in the United States is done in a completely different way in Italy. The first time I drove in Italy, I thought the Italian drivers were just giving me a friendly wave.

The situation is no different when a human and a computer communicate. The computer would probably be most efficient if you could converse with it in simple binary, but you're most comfortable if the computer uses words and pictures that you can readily understand. Fortunately, the computer has no vote in this matter. It's up to the programmer to instruct the machine to use appropriate human communication techniques.

In most cases, the human-computer dialog is conducted in the human's written language and with the help of some relatively simple pictures such as icons, mathematical charts, or musical scores. These pictures tend to be locale independent, except for words that might be embedded within them. Often the embedded words need no translation because they're part of an international jargon associated with the picture. For instance, musical scores use certain Italian words that are recognized by musicians throughout the world. Similarly, our international community of computer specialists uses a subset of English for Basic, C, C++, Java, Fortran, Pascal, Cobol, RPG, and other programming languages. As I mentioned earlier, this is why many of us mistakenly assume that computer users are also familiar with English.

Some computer applications employ more complex forms of communication such as voice recognition and synthesis or expressive graphics of the "Max Headroom" variety. This book won't describe these forms of communication because relatively few computer users and programmers are currently exposed to them. Furthermore, complex aural and visual communication techniques are still more theory than practice, and the technology is quite expensive. Fortunately, these forms of communication are extremely country dependent, and so the scientists and engineers evolving this branch of computing are already thinking of international solutions. For example, imagine how shortsighted it would be for a car manufacturer to incorporate a computerized verbal warning system that could express itself only in Swedish.

This book concentrates on the more prevalent written forms of human-computer communication, which are implemented by means of keyboards, display screens, and printers. This book also focuses on desktop computers rather than minicomputers or mainframes because most major new human interface work is being done on the various personal computer platforms.

Furthermore, this book examines cosmopolitan software examples that are based on the Microsoft Win32 operating system model and the C++ programming language because they have the most complete set of international programming features that I've found.

Before leaving this topic, I must point out that some software can be chauvinistic even though it has little or no human interface. For example, consider a program that's invoked through a simple command line to sort a file in alphabetical order. This program must cope with the fact that different countries use different alphabets. It can't simply sort characters by their numeric codes, since that usually works only with the English alphabet. This and other subtle locale dependencies are covered in more detail when I describe character sets.

Adaptation Methods

Preparing software for use in a particular country is a two-step process. First the designer must enable the program for international usage. Then someone—probably not the original designer—must adapt the program to each country in which it will be used. In some ways, this enabling and adapting process is similar to the two-step process used in the international book publishing business. The original author writes the book in his or her native language, and then other writers translate it into various languages, depending on where the publisher decides to sell the book.

All of the techniques and guidelines presented in this book are concerned with the enabling step, since that's our primary concern as software designers. We need to understand the adaptation step, however, because it can become a major cost item, especially if we choose the wrong enabling technique.

For example, you can enable a program by isolating all of the prompters and messages and putting them into header files that are compiled with the source files. In this case, the adaptation step requires that someone edit or replace the header files, compile all of the program source, and link the object files to produce a new executable file. This work is best done by another programmer who can handle the inevitable compiler error messages and can do at least a small amount of regression testing to ensure that the new executable works correctly.

Suppose instead that you enable the program by placing prompters and messages into a data file processed during the program's initialization phase. In that case, a nonprogrammer can supply the appropriate data file for his or her country and can easily verify that the program operates correctly with that data. Generally, this second technique makes the adaptation step cheaper and less prone to introducing program errors.

I've found that there's one simple but important question you must ask before choosing a particular enabling technique: *Who will be adapting this software for use in other countries?* The possibilities are

- The original development team
- Another development team
- A professional translation group
- The end user
- The local software distributor

No single answer is the only correct one because teams, projects, products, and markets have so many differences. Furthermore, the adaptation step can be spread out among several organizations, and this often causes the enabling technique to change as the program evolves. Early adaptations for the most important markets are handled by highly skilled people, possibly other programmers, in order to work out the kinks in the enabling technology. Then the later adaptations are handled by less skilled (and less expensive) translators.

For instance, I once worked on a project in which the programming team included a person fluent in German. We used message files to enable the software for international use and then tested this technique by actually doing the German adaptation from the original English. Our major European distributor later adapted the program for the French and Italian markets, while some clever users did the work for Spanish, Flemish, Swedish, and several other languages. In the second version of the software, we gathered all of these adaptations into a library and changed the installation procedure so that the end user could simply choose the appropriate language.

Let's consider the merits of these adaptation methods. Then I'll explain which enabling techniques are appropriate for them.

Adaptation by the Original Development Team

In some ways, the original development team is ideally suited to handle the adaptation of a software package to other languages. They have the knowledge and the tools to build the program from its source code and can deal with language differences at that level, which usually results in the smallest and fastest