LẬP TRÌNH C TRÊN WINDOWS

CHƯƠNG I: TỔNG QUAN LẬP TRÌNH C TRÊN WINDOWS

A. MỞ ĐẦU

Để lập trình trên Microsoft Windows®, chúng ta cần nắm được các đặc điểm cơ bản nhất của hệ điều hành này. Chương này sẽ giới thiệu khái quát các đặc điểm hệ điều hành Microsoft Windows, các vấn đề liên quan đến lập trình bằng ngôn ngữ C, đồng thời đưa ra một chương trình mẫu làm sườn cho các chương trình được viết sau này.

Trong phần đầu, chúng ta tìm hiểu sơ lược lịch sử phát triển của hệ điều hành Microsoft Windows® và những đặc điểm nền tảng của Windows.

Phần tiếp theo sẽ trình bày những khái niệm và yêu cầu căn bản của việc lập trình C trên Windows. Ngoài ra, phần này cũng giới thiệu các cơ chế và các công cụ mà hệ điều hành cung cấp cho người lập trình hay người phát triển các ứng dụng trên Windows.

Cuối chương là phần xây dựng một chương trình đơn giản nhất trên Windows. Chương trình này được xem như là khuôn mẫu của một chương trình ứng dụng điển hình, và hầu hết các đoạn chương trình được viết minh họa trong sách đều lấy chương trình này làm khung sườn để phát triển cho phù hợp với từng yêu cầu. Thêm vào đó, một số kiểu dữ liệu mới được định nghĩa trên Windows và những qui ước về cách đặt tên biến cũng được giới thiệu trong phần này.

Phần chi tiết và chuyên sâu hơn của việc lập trình bằng ngôn ngữ C trên môi trường Windows sẽ được trình bày trong các chương tiếp theo.

B. HỆ ĐIỀU HÀNH MICROSOFT WINDOWS

1. Giới thiệu

Giữa thập niên 80, công ty phần mềm máy tính Microsoft công bố phiên bản đầu tiên của dòng hệ điều hành Windows là **Microsoft Windows®** 1.0. Đây là hệ điều hành dùng giao diện đồ họa khác với giao diện ký tự (*text* hay *console*) của **MS-DOS**. Tuy nhiên phải đến phiên bản thứ hai (Windows 2.0 - tháng 11 năm 1987) thì mới có bước cải tiến đáng kể, đó là sự mở rộng giao tiếp giữa bàn phím và thiết bị chuột và giao diện đồ họa (**GUI**-*Graphic User Interface*) như trình đơn (*menu*) và hộp thoại (*dialog*). Trong phiên bản này Windows chỉ yêu cầu bộ vi xử lý Intel 8086 hay 8088 chạy ở real-mode để truy xuất 1 megabyte bộ nhớ.

Tháng 5 năm 1990, Microsoft công bố phiên bản tiếp theo là Windows 3.0. Sự thay đổi lớn trong phiên bản này là Windows 3.0 hỗ trợ protected-mode 16 bit của các bộ vi xử lý 286, 386, và 486 của Intel. Sự thay đổi này cho phép các ứng dụng trên Windows truy xuất 16 megabyte bộ nhớ. Tiếp bước với sự phát triển là phiên bản Windows 3.1 ra đời năm 1992, Microsoft đưa công nghệ Font TrueType, âm nhạc (*multimedia*), liên kết và nhúng đối tượng (**OLE**- *Object Linking and Embedding*), và đưa ra các hộp thoại chung đã được chuẩn hóa.

Trong sự phát triển mạnh mẽ của những thập niên 90, Microsoft công bố tiếp dòng hệ điều hành Windows với ứng dụng công nghệ mới (1993). Hệ điều hành này lấy tên là **Windows® NT®** (Windows *New Technology*), đây là phiên bản hệ điều hành đầu tiên của Windows hỗ trợ 32 bit cho bộ xử lý 386, 486 và Pentium. Trong hệ điều hành này thì các ứng dụng phải truy xuất bộ nhớ với địa chỉ là 32-bit và các tập lệnh hay chỉ thị 32-bit. Ngoài ra Windows NT cũng được thiết kế để chạy các bộ vi xử lý (*CPU*) khác ngoài Intel và có thể chạy trên các máy trạm (*workstation*).

Hệ điều hành Windows 95 được công bố năm 1995 cũng là một hệ điều hành 32-bit cho Intel 386 trở về sau. Tuy thiếu tính bảo mật như Windows NT và việc thích nghi với máy trạm công nghệ **RISC**, nhưng bù lại hệ điều hành này yêu cầu phần cứng không cao.

Song song với sự phát triển phần mềm thì công nghệ phần cứng cũng phát triển không kém. Để tận dụng sức mạnh của phần cứng thì các thế hệ Windows tiếp theo ngày càng hoàn thiện hơn. Như Windows 98 phát triển từ Window 95 và có nhiều cải thiện như hiệu năng làm việc, hỗ trợ các thiết bị phần cứng tốt hơn, và cuối cùng là việc tích hợp chặt chẽ với **Internet** và **Word Wide Web**.

Windows 2000 là hệ điều hành được xem là ổn định và tốt của dòng Windows, phiên bản này tăng cường các tính năng bảo mật thích hợp trong mội trường mạng và giao diện đẹp.

2. Đặc điểm chung của hệ điều hành Microsoft Windows

Windows là một hệ điều hành sử dụng giao tiếp người dùng đồ họa (GUI), hay còn gọi là hệ điều hành trực quan (*Visual interface*). **GUI** sử dụng đồ họa dựa trên màn hình ảnh nhị phân (*Bitmapped video display*). Do đó tận dụng được tài nguyên thực của màn hình, và cung cấp một môi trường giàu tính trực quan và sinh động.

Windows không đơn điệu như **MS-DOS** (hay một số hệ điều hành giao diện console) mà màn hình được sử dụng chỉ để thể hiện chuỗi ký tự, do người dùng gõ từ bàn phím (*keyboard*) hay để xuất thông tin dạng văn bản. Trong giao diện người dùng đồ họa, màn hình giao tiếp với người sử dụng đa dạng hơn, người dùng có thể nhập dữ liệu thông qua chuột bằng cách nhấn vào các nút nhấn (*button*) các hôp chọn (*combo box*)…thiết bị bây giờ được nhập, có thể là bàn phím và thiết bị chuột (*mouse device*). Thiết bị chuột là một thiết bị định vị trên màn hình, sử dụng thiết bị chuột người dùng có thể nhập dữ liệu một cách trực quan bằng cách kích hoạt một nút lệnh, hay làm việc với các đối tượng đồ họa liên quan đến tọa độ trên màn hình.

Để giao tiếp trong môi trường đồ họa, Windows đưa ra một số các thành phần gọi là các điều khiển chung (*common control*), các điều khiển chung là các đối tượng được đưa vào trong hộp thoại để giao tiếp với người dùng. Bao gồm: hộp văn bản (*text box*), nút nhấn (*button*), nút chọn (*check box*), hộp danh sách (*list box*), hộp chọn (*combo box*)…

Thật ra một ứng dụng trên Windows không phải là quá phức tạp vì chúng có hình thức chung. Chương trình ứng dụng thuờng chiếm một phạm vi hình chữ nhật trên màn hình gọi là một cửa sổ. Trên cùng của mỗi cửa sổ là thanh tiêu đề (*title bar*). Các chức năng của chương trình thì được liệt kê trong thực đơn lựa chọn của chương trình (*menu*), hay xuất hiện dưới dạng trực quan hơn là các thanh công cụ (*toolbar*). Các thanh công cụ này chứa các chức năng được sử dụng thường xuyên trong thực đơn để giảm thời gian cho người dùng phải mở thực đơn và chọn. Thông thường khi cần lấy thông tin hay cung cấp thông tin cho người dùng thì một ứng dụng sẽ đưa ra một hộp thoại, trong hộp thoại này sẽ chứa các điều khiển chung để giao tiếp với người dùng. Windows cũng ra tạo một số các hộp thoại chuẩn như **Open Files**, và một số hộp thoại tương tự như nhau.

Windows là một hệ điều hành đa nhiệm, tùy thuộc vào bộ nhớ mà ta có thể chạy nhiều ứng dụng cùng một lúc, và cũng có thể đồng thời chuyển qua lại giữa các ứng dụng và thực thi chúng. Trong các phiên bản của Windows® 98 và NT® trở về sau, các chương trình ứng dụng tự bản thân chúng chia thành nhiều tiểu trình (*thread*) để xử lý và với tốc độ xử lý nhanh tạo cảm giác những chương trình ứng dụng này chạy đồng thời với nhau.

Trong Windows, chương trình ứng dụng khi thực thi được chia sẻ những thủ tục mà Windows cung cấp sẵn, các tập tin cung cấp những thủ tục trên được gọi là thư viện liên kết động (*Dynamic Link Libraries* - **DLL**). Windows có cơ chế liên kết những chương trình ứng dụng với các thủ tực được cung cấp trong thư viện liên kết động.

Khả năng tương thích của Windows cũng rất cao. Các chương trình ứng dụng được viết cho Windows không truy xuất trực tiếp phần cứng của những thiết bị đồ hoạ như màn hình và máy in. Mà thay vào đó, hệ điều hành cung cấp một ngôn ngữ lập trình đồ họa (gọi là *Giao tiếp thiết bị đồ hoạ - Graphic Device Interface* - **GDI**) cho phép hiển thị những đối tượng đồ họa một cách dễ dàng. Nhờ vậy một ứng dụng viết cho Windows sẽ chạy với bất cứ thiết bị màn hình nào hay bất kì máy in, miễn là đã cài đặt trình điều khiển thiết bị hỗ trợ cho Windows. Chương trình ứng dụng không quan tâm đến kiểu thiết bị kết nối với hệ thống.

Như giới thiệu ở phần trên khái niệm liên kết động là thành phần quan trọng của Windows, nó được xem như là hạt nhân của hệ điều hành, vì bản thân của Windows là các tập thư viện liên kết động. Windows cung cấp rất nhiều hàm cho những chương trình ứng dụng để cài đặt giao diện người dùng và hiển thị văn bản hay đồ họa trên màn hình. Những hàm này được cài đặt trong thư viện liên kết động hay còn gọi là **DLL**. Đó là các tập tin có dạng phần mở rộng là ***.DLL** hay ***.EXE**, hầu hết được chứa trong thư mục **\Windows\System**, **\Windows\system32** của **Windows® 98** và các thư mục **\WinNT\System**, **\WinNT\System32** của **Windows® NT®**.

Trong các phiên bản sau này, hệ thống liên kết động được tạo ra rất nhiều, tuy nhiên, hầu hết các hàm được gọi trong thư viện này phân thành 3 đơn vị sau: ***Kernel***, ***User***, và ***GDI***.

Kernel cung cấp các hàm và thủ tục mà một hạt nhân hệ điều hành truyền thống quản lý, như quản lý bộ nhớ, xuất nhập tập tin và tác vụ. Thư viện này được cài đặt trong tập tin **KRNL386.EXE** 16 bit và **KERNEL32.DLL** 32 bit.

User quản lý giao diện người dùng, cài đặt tất cả khung cửa sổ ở mức luận lý. Thư viện User được cài đặt trong tập tin **USER.EXE** 16 bit và **USER32.DLL** 32 bit.

GDI cung cấp toàn bộ giao diện thiết bị đồ hoạ (*Graphics Device Interface*), cho phép chương trình ứng dụng hiển thị văn bản và đồ hoạ trên các thiết bị xuất phần cứng như màn hình và máy in.

Trong Windows 98, thư viện liên kết động chứa khoảng vài ngàn hàm, mỗi hàm có tên đặc tả, ví dụ **CreateWindow**, hàm này dùng để tạo một cửa sổ cho ứng dụng. Khi sử dụng các hàm mà Windows cung cấp cho thì các ứng dụng phải khai báo trong các tập tin tiêu đề.**h** hay.**hpp** (*header file*).

Trong một chương trình Windows, có sự khác biệt khi ta gọi một hàm của thư viện C và một hàm của Windows hay thư viện liên kết động cung cấp. Đó là khi biên dịch mã máy, các hàm thư viện C sẽ được liên kết thành mã chương trình. Trong khi các hàm Windows sẽ được gọi khi chương trình cần dùng đến chứ không liên kết vào chương trình. Để thực hiện được các lời gọi này thì một chương trình Windows *.EXE luôn chứa một tham chiếu đến thư viện liên kết động khác mà nó cần dùng. Khi đó, một chương trình Windows được nạp vào bộ nhớ sẽ tạo con trỏ tham chiếu đến những hàm thư viện DLL mà chương trình dùng, nếu thư viện này chưa được nạp vào bộ nhớ trước đó thì bây giờ sẽ được nạp.

C. LẬP TRÌNH TRÊN MICROSOFT WINDOWS

1. Đặc điểm chung

Windows là hệ điều hành đồ họa trực quan, do đó các tài nguyên của hệ thống cung cấp rất đa dạng đòi hỏi người lập trình phải nghiên cứu rất nhiều để phát huy hết sức mạnh của hệ điều hành.

Theo như những mục đích tiếp cận của các nhà lập trình thì các ứng dụng trên Windows phải hết sức thân thiện với người dùng thông qua giao diện đồ họa sẵn có của Windows. Về lý thuyết thì một người dùng làm việc được với một ứng dụng của Windows thì có thể làm việc được với những ứng dụng khác. Nhưng trong thực tế để sử dụng một ứng dụng cho đạt hiệu quả cao trong Windows thì cần phải có một số huấn luyện trợ giúp hay tối thiểu thì phải cho biết chương trình ứng dụng làm việc như thế nào.

Đa số các ứng dụng trong Windows đều có chung một giao diện tương tác với người dùng giống nhau. Ví dụ như các ứng dụng trong Windows đa số đều có thanh thực đơn chứa các mục như: **File**, **Edit**, **Tool**, **Help**… Và trong hộp thoại thì thường chứa các phần tử điều khiển chung như: **Edit Control**, **Button Control**, **Checkbox**….

2. Sự khác biệt với lập trình trên MS-DOS

Khi mới bước vào lập trình trên Windows đa số người học rất lạ lẫm, nhất là những người đã từng làm việc với MS-DOS. Do MS-DOS là hệ điều hành đơn nhiệm và giao tiếp qua giao diện console. Nên khi viết chương trình không phức tạp.

Còn đối với Windows người lập trình sẽ làm việc với bộ công cụ lập trình đồ họa đa dạng cùng với cách xử lý đa nhiệm, đa luồng của Windows. Vì vậy việc lập trình trên Windows sẽ giúp cho người lập trình đỡ nhàm chán với giao diện console của MS-DOS. Việc cố gắng phát huy các sức mạnh tài nguyên của Windows sẽ làm cho những ứng dụng càng mạnh mẽ, đa dạng, thân thiện, và dễ sử dụng.

3. Một số yêu cầu đối với người lập trình

Điều trước tiên của người học lập trình C trên Windows là phải biết lập trình C, sách này không có tham vọng hướng dẫn người học có thể thông thạo lập trình C trên Windows mà chưa qua một lớp huấn luyện C nào. Tuy nhiên, không nhất thiết phải hoàn toàn thông thạo C mới học được lập trình Windows.

Để có thể lập trình trên nền Windows ngoài yêu cầu về việc sử dụng công cụ lập trình, người học còn cần phải có căn bản về Windows, tối thiểu thì cũng đã dùng qua một số ứng dụng trong Windows. Thật sự yêu cầu này không quá khó khăn đối với người học vì hiện tại hầu như Windows quá quen thuộc với mọi người, những người mà đã sử dụng máy tính.

Ngoài những yêu cầu trên, đôi khi người lập trình trên Windows cũng cần có khiếu thẩm mỹ, vì cách trình bày các hình ảnh, các điều khiển trên các hộp thoại tốt thì sẽ làm cho ứng dụng càng tiện lợi, rõ ràng, và thân thiện với người dùng.

4. Bộ công cụ giao diện lập trình ứng dụng API

Hệ điều hành Windows cung cấp hàng trăm hàm để cho những ứng dụng có thể sử dụng truy cập các tài nguyên trong hệ thống. Những hàm này được gọi là giao diện lập trình ứng dụng **API** (*Application Programming Interface*). Những hàm trên được chứa trong các thư viện liên kết động **DLL** của hệ thống. Nhờ có cấu trúc động này mọi ứng dụng đều có thể truy cập đến các hàm đó. Khi biên dịch chương trình, đến đoạn mã gọi hàm API thì chương trình dịch không thêm mã hàm này vào mã thực thi mà chỉ thêm tên DLL chứa hàm và tên của chính hàm đó. Do đó mã các hàm API thực tế không được sử dụng khi xây dựng chương trình, và nó chỉ được thêm vào khi chương trình được nạp vào bộ nhớ để thực thi.

Trong API có một số hàm có chức năng duy trì sự độc lập thiết bị đồ họa, và các hàm này gọi là giao diện thiết bị đồ họa **GDI** (*Graphics Device Interface*). Do sự độc lập thiết bị nên các hàm **GDI** cho phép các ứng dụng có thể làm việc tốt với nhiều kiểu thiết bị đồ họa khác nhau.

5. Cơ chế thông điệp

Không giống như các ứng dụng chạy trên MS-DOS, các ứng dụng Win32® thì xử lý theo các sự kiện (*event - driven*), theo cơ chế này các ứng dụng khi được viết sẽ liên tục chờ cho hệ điều hành truyền các dữ liệu nhập vào. Hệ thống sẽ đảm nhiệm việc truyền tất cả các dữ liệu nhập của ứng dụng vào các cửa sổ khác nhau của ứng dụng đó. Mỗi một cửa sổ sẽ có riêng một hàm gọi là hàm xử lý cửa sổ thường được đặt tên là **WndProc**, hệ thống sẽ gọi hàm này khi có bất cứ dữ liệu nhập nào được truyền đến cửa sổ, hàm này sẽ xử lý các dữ liệu nhập đó và trả quyền điều khiển về cho hệ thống.

Hệ thống truyền các dữ liệu nhập vào thủ tục xử lý của cửa sổ thông qua một hình thức gọi là thông điệp (*message*). Thông điệp này được phát sinh từ ứng dụng và hệ thống. Hệ thống sẽ phát sinh một thông điệp khi có một sự kiện nhập vào (*input even*), ví dụ như khi người dùng nhấn một phím, di chuyển thiết bị chuột, hay kích vào các điều khiển (*control*) như thanh cuộn,… Ngoài ra hệ thống cũng phát sinh ra thông

điệp để phản ứng lại một sự thay đổi của hệ thống do một ứng dụng mang đến, điều này xảy ra khi ứng dụng làm cạn kiệt tài nguyên hay ứng dụng tự thay đổi kích thước của cửa sổ.

Một ứng dụng có thể phát sinh ra thông điệp khi cần yêu cầu các cửa sổ của nó thực hiện một nhiệm vụ nào đó hay dùng để thông tin giữa các cửa sổ.

Hệ thống gởi thông điệp vào thủ tục xử lý cửa sổ với bốn tham số: định danh của cửa sổ, định danh của thông điệp, và hai tham số còn lại được gọi là tham số của thông điệp (*message parameters*). Định danh của cửa sổ xác định cửa sổ mà thông điệp được chỉ định. Hệ thống sẽ dùng định danh này để xác định cần phải gởi thông điệp đến thủ tục xử lý của cửa sổ.

Định danh thông điệp là một hằng số thể hiện mục đích của thông điệp. Khi thủ tục xử lý cửa sổ nhận thông điệp thì nó sẽ dùng định danh này để biết hình thức cần thực hiện. Ví dụ, khi một thông điệp được truyền đến thủ tục cửa sổ có định danh là **WM_PAINT** thì có ý nghĩa rằng cửa sổ vùng làm việc thay đổi và cần phải vẽ lại vùng này.

Tham số thông điệp lưu giá trị hay vị trí của dữ liệu, được dùng bởi thủ tục cửa sổ khi xử lý thông điệp. Tham số này phụ thuộc vào loại thông điệp được truyền đến, nó có thể là số nguyên, một tập các bit dùng làm cờ hiệu, hay một con trỏ đến một cấu trúc dữ liệu nào đó,…

Khi một thông điệp không cần dùng đến tham số thì hệ thống sẽ thiết lập các tham số này có giá trị NULL. Một thủ tục cửa sổ phải kiểm tra xem với loại thông điệp nào cần dùng tham số để quyết định cách sử dụng các tham số này.

Có hai loại thông điệp:

Thông điệp được định nghĩa bởi hệ thống (*system-defined messages*):

Dạng thông điệp này được hệ thống định nghĩa cho các cửa sổ, các điều khiển, và các tài nguyên khác trong hệ thống. Thường được bắt đầu với các tiền tố sau: **WM_xxx**, **LB_xxx**, **CB_xxx**,…

Thông điệp được định nghĩa bởi ứng dụng (*application-defined message*):

Một ứng dụng có thể tạo riêng các thông điệp để sử dụng bởi những cửa sổ của nó hay truyền thông tin giữa các cửa sổ trong ứng dụng.

Nếu một ứng dụng định nghĩa các thông điệp riêng thì thủ tục cửa sổ nhận được thông điệp này phải cung cấp các hàm xử lý tương ứng.

Đối với thông điệp hệ thống, thì được cung cấp giá trị định danh từ **0x0000** đến **0x03FF**, những ứng dụng không được định nghĩa thông điệp có giá trị trong khoảng này.

Thông điệp được ứng dụng định nghĩa có giá trị định danh từ **0x0400** đến **0x7FFF**.

Lộ trình của thông điệp từ lúc gởi đi đến lúc xử lý có hai dạng sau:

Thông điệp được gởi vào hàng đợi thông điệp để chờ xử lý (*queue message*): bao gồm các kiểu thông điệp được phát sinh từ bàn phím, chuột như thông điệp: **WM_MOUSEMOVE**, **WM_LBUTTONDOWN**, **WM_KEYDOWN,** và **WM_CHAR**.

Thông điệp được gởi trực tiếp đến thủ tục xử lý không qua hàng đợi (*nonqueue message*), bao gồm các thông điệp thời gian, thông điệp vẽ, và thông điệp thoát như **WM_TIMER**, **WM_PAINT**, và **WM_QUIT**.

Xử lý thông điệp: Một ứng dụng phải xóa và xử lý những thông điệp được gởi tới hàng đợi của ứng dụng đó. Đối với một ứng dụng đơn tiểu trình thì sử dụng một vòng lặp thông điệp (*message loop*) trong hàm **WinMain** để nhận thông điệp từ hàng đợi và gởi tới thủ tục xử lý cửa sổ tương ứng. Với những ứng dụng nhiều tiểu trình thì mỗi một tiểu trình có tạo cửa sổ thì sẽ có một vòng lặp thông điệp để xử lý thông điệp của những cửa sổ trong tiểu trình đó.

D. CÁCH VIẾT MỘT ỨNG DỤNG TRÊN MICROSOFT WINDOWS

1. Các thành phần cơ bản tạo nên một ứng dụng

a. Cửa sổ

Trong một ứng dụng đồ họa 32-bit, cửa sổ (*window*) là một vùng hình chữ nhật trên màn hình, nơi mà ứng dụng có thể hiển thị thông tin ra và nhận thông tin vào từ người sử dụng. Do vậy, nhiệm vụ đầu tiên của một ứng dụng đồ họa 32-bit là tạo một cửa sổ.

Một cửa sổ sẽ chia sẻ màn hình với các cửa sổ khác trong cùng một ứng dụng hay các ứng dụng khác. Chỉ một cửa sổ trong một thời điểm nhận được thông tin nhập từ người dùng. Người sử dụng có thể dùng bàn phím, thiết bị chuột hay các thiết bị nhập liệu khác để tương tác với cửa sổ và ứng dụng.

Tất cả các cửa sổ đều được tạo từ một cấu trúc được cung cấp sẵn gọi là lớp cửa sổ (*window class*). Cấu trúc này là một tập mô tả các thuộc tính mà hệ thống dùng như khuôn mẫu để tạo nên các cửa sổ. Mỗi một cửa sổ phải là thành viên của một lớp cửa sổ. Tất cả các lớp cửa sổ này đều được xử lý riêng biệt.

b. Hộp thoại và các điều khiển

Hộp thoại (*Dialog*) dùng để tương tác với người dùng trong một chương trình ứng dụng. Một hộp thoại thường chứa nhiều các đều khiển như ô nhập văn bản (*edit text*), nút bấm (*button*), ghi chú (*static control*), hộp danh sách (*list box*)…

- **Nút bấm** *(button)*: gồm có **Push Button** dùng kích hoạt một thao tác, **Check Box** dùng để chọn một trong hai trạng thái (TRUE hay FALSE), **Radio Button** cũng giống như Check Box nhưng một nhóm các Radio Button phải được chọn loại trừ nhau.
- **Chú thích** (*static*): dùng để chứa các ghi chú trong hộp thoại, ngoài ra nội dung có thể thay đổi trong quá trình sử dụng hộp thoại.
- **Hộp liệt kê** (*list box*): Chọn một hay nhiều dữ liệu được liệt kê trong danh sách, nếu hộp chứa nhiều dòng và hộp không hiển thị hết các mẫu thông tin thì phải kèm theo một thanh cuộn (*scroll bar*).
- **Ô nhập văn bản** (*edit text*): Dùng nhập văn bản, nếu ô có nhiều dòng thì thường kèm theo thanh cuộn.
- **Thanh cuộn** (*scroll bar*): ngoài việc dùng kèm với list box hay edit box thì thanh cuộn còn có thể sử dụng độc lập nhằm tạo các thước đo…
- **Thực đơn** (*menu*): là một danh sách chứa các thao tác với một định danh mà người dùng có thể chọn. Hầu hết các ứng dụng có cửa sổ thì không thể thiếu thực đơn.
- **Thanh công cụ** (*toolbar*): đây là một dạng menu nhưng chỉ chứa các thao tác cần thiết dưới dạng các biểu tượng đặc trưng.

  Ngoài ra còn rất nhiều các điều khiển mà các công cụ lập trình cung cấp cho người lập trình hay tự họ tạo ra dựa trên những thành phần được cung cấp sẵn.

c. Ứng dụng điển hình trên Windows

d. Các kiểu tập tin để xây dựng một ứng dụng trên Windows

Chương trình nguồn

Tương tự như các chương trình C chuẩn, bao gồm các tập tin tiêu đề (*header*) chứa trong tập tin ***.h**, ***.hpp**. Còn mã nguồn (*source code*) chứa trong tập tin ***.c** hay ***.cpp**.

Tập tin định nghĩa

Tập tin này có phần mở rộng là ***.def**, dùng định nghĩa các điều khiển do chương trình tạo ra khi viết ứng dụng tạo **DLL**, ngoài ra còn dùng để khai báo vùng nhớ heap khi chạy chương trình. Lúc trước do vấn đề tương thích với Windows 3.1 nên tập tin này thường được dùng, còn ngày nay chúng ít được dùng đến.

Các file chứa tài nguyên của ứng dụng

- Các file ***.ico** là các biểu tượng (*icon*) được dùng trong chương trình. Thông thường các công cụ lập trình trên Windows đều có các tool để tạo các ảnh này.
- Con trỏ chuột của ứng dụng có thể được vẽ lại dưới dạng các biểu tượng và lưu trên đĩa với dạng file ***.cur**.
- Các file dạng ảnh bitmap dùng để minh họa được lưu dạng file ***.bmp.**
- Tập tin tài nguyên ***.rc** là phần khai báo các tài nguyên như thực đơn, hộp thoại, và các định danh chỉ đến các tập tin dạng ***.ico**, ***.cur**, ***.bmp**,...

e. Các kiểu dữ liệu mới

Các kiểu dữ liệu trên Windows thường được định nghĩa nhờ từ khoá **`typedef`** trong tập tin **windows.h** hay các tập tin khác. Thông thường các tập tin định nghĩa này do Microsoft viết ra hoặc các công ty viết trình biên dịch C tạo ra, nhất thiết nó phải tương thích với hệ điều hành Windows 98, hay NT dựa trên kiến trúc 32-bit.

Một vài kiểu dữ liệu mới có tên viết tắt rất dễ hiểu như **`UINT`** là một dữ liệu thường được dùng mà đơn giản là kiểu **`unsigned int`**, trong Windows 9x kiểu này có kích thước là 32-bit. Đối với kiểu chuỗi thì có kiểu **`PSTR`** kiểu này là một con trỏ đến một chuỗi tương tự như **`char*`**.

Tuy nhiên, cũng có một số kiểu được khái báo tên thiếu rõ ràng như **`WPARAM`** và **`LPARAM`**. Tên này được đặt vì có nguồn ngốc lịch sử sâu xa. Khi còn hệ điều hành Windows 16-bit thì tham số thứ 3 của hàm **`WndProc`** được khai báo là kiểu **`WORD`**, với kích thước 16-bit, còn tham số thứ 4 có kiểu **`LONG`** là 32-bit. Đây là lý do người ta thêm tiến tố "**`W`**", "**`L`**" vào từ "**`PARAM`**". Tuy nhiên, trong phiên bản Windows 32-bit, thì **`WPARAM`** được định nghĩa như là **`UINT`** và **`LPARAM`** thì được định nghĩa như một kiểu **`LONG`**, do đó cả hai tham số này đều có giá trị là 32-bit. Điều này là một sự nhầm lẫn vì **`WORD`** vẫn là giá trị 16-bit trong Window 98.

Trong thủ tục xử lý cửa sổ **`WndProc`** giá trị trả về là kiểu **`LRESULT`**. Kiểu này đơn giản được định nghĩa như là kiểu **`LONG`**.

Ngoài ra, có một kiểu thường xuyên dùng là kiểu **`HANDLE`** là một số nguyên 32-bit được sử dụng như một kiểu định danh. Có nhiều kiểu định danh nhưng nhất thiết tất cả phải có cùng kích thước với **`HANDLE`**.

Bảng sau mô tả một số kiểu dữ liệu mới:

Kiểu	Ý nghĩa
HANDLE	Số nguyên 32-bit, định danh.
HWND	Số nguyên 32-bit, định danh.

BYTE	Giá trị 8-bit không dấu.
WORD	Số nguyên 16-bit không dấu.
DWORD	Số nguyên 32-bit không dấu.
UINT	Số nguyên không dấu 32-bit.
LONG	long 32-bit.
BOOL	Bool.
LPSTR	Con trỏ chuỗi.
LPCSTR	Hằng con trỏ chuỗi.
WPARAM	32-bit.
LPARAM	32-bit.
BSTR	Giá trị 32-bit trỏ đến kí tự.
LPVOID	Con trỏ 32-bit đến một kiểu không xác định.
LPTSTR	Giống như LPSTR nhưng có thể chuyển sang dạng Unicode và DBCS.
LPCTSTR	Giống như LPCSTR nhưng có thể chuyển sang dạng Unicode và DBCS.

Bảng 1.1 Mô tả các kiểu dữ liệu mới

2. Khuôn mẫu chung tạo một ứng dụng

Một ứng dụng đơn giản nhất của Windows bao gồm có hai hàm là **WinMain** và xử lý cửa sổ **WinProc**. Do đó hai hàm này là quan trọng và không thể thiếu trong các ứng dụng Windows.

Hàm **WinMain** thực hiện các chức năng sau:

- Định nghĩa lớp cửa sổ ứng dụng.
- Đăng ký lớp cửa sổ vừa định nghĩa.
- Tạo ra thể hiện cửa sổ của lớp đã cho.
- Hiển thị cửa sổ.
- Khởi động chu trình xử lý thông điệp.

Hàm xử lý **WinProc** có chức năng xử lý tất cả các thông điệp có liên quan đến cửa sổ.

3. Hàm WinMain

Hàm chính của một ứng dụng chạy trên Windows là hàm **WinMain**, được khai báo như sau:

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   PSTR szCmdLine, int iCmdShow)
```

Chúng ta sẽ tìm hiểu một hàm **WinMain** mẫu sau đây.

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   PSTR szCmdLine, int iCmdShow)
{
    static TCHAR szAppName[] = TEXT("HelloWin"); // tên ứng dụng
    HWND hwnd;
    MSG msg;
    WNDCLASS wndclass; // biến để định danh một cửa sổ
    /* Định nghĩa kiểu cửa sổ */
    wndclass.style = SC_HREDRAW | CS_VREDRAW;
    wndclass.lpfnWndProc = WndProc; // Hàm thủ tục cửa sổ
    wndclass.cbClsExtra = 0;
    wndclass.cbWndExtra = 0;
    wndclass.hInstance = hInstance; // Định danh ứng dụng
    wndclass.hIcon = LoadIcon (NULL, IDI_APPLICATION);
    wndclass.hCursor = LoadCusor (NULL, IDC ARROW);
    wndclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wndclass.lpszMenuName = NULL; // Không có menu
    wndclass.lpszClassName = szAppName; // tên ứng dụng
    /* Đăng ký lớp cửa sổ */
    if (!RegisterClass(&wndclass)) return 0;
    /* Tạo lớp cửa sổ */
    hwnd = CreateWindow(szAppName, // Tên cửa sổ
                        "Hello Program", // Tiêu đề
                        WS_OVERLAPPEDWINDOW, // Kiểu cửa sổ
```

```
                        CW_USEDEFAULT, // Toạ độ x
                        CW_USEDEFAULT, // Toạ độ y
                        CW_USEDEFAULT, // Chiều rộng
                        CW_USEDEFAULT, // Chiều dài
                        NULL, // Cửa sổ cha
                        NULL, // Không có menu
                        hInstacne, // Định danh ứng dụng
                        NULL); // Tham số bổ sung
    /* Hiển thị cửa sổ */
    ShowWindow (hwnd, iCmdShow);
    UpdateWindow (hwnd);
    /* Chu trình xử lý các thông điệp*/
    while (GetMessage (&msg, NULL, 0, 0)) {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }
    return msg.wParam;
}
```

Định nghĩa một lớp cửa sổ:

Đầu tiên của việc xây dựng một ứng dụng Windows là phải định nghĩa một lớp cửa sổ cho ứng dụng. Windows cung cấp một cấu trúc **WNDCLASS** gọi là lớp cửa sổ, lớp này chứa những thuộc tính tạo thành một cửa sổ.

```
typedef struct _WNDCLASS
{
    UINT style;
    WNDPROC lpfnWndProc;
    int cbClsExtra;
    HINSTANCE hInstance;
    HICON hIcon;
    HCURSOR hCursor;
    HBRUSH hbrBackground;
    LPCSTR lpszMenuName;
    LPCSTR lpszClassName;
} WNDCLASS, *PWNDCLASS;
```

Ý nghĩa thuộc tính của cấu trúc **WNDCLASS** được mô tả trong bảng sau:

Thuộc tính	Ý nghĩa	Ghi chú
style	Kiểu lớp	Kết hợp nhiều kiểu giá trị khác nhau bằng toán tử OR.
lpfnWndProc	Con trỏ đến thủ tục window	
cbClsExtra	Số byte được cấp phát thêm sau cấu trúc window-class	Mặc định
cbWndExtra	Số byte được cấp phát thêm sau một instance của window	Mặc định
hInstance	Định danh chứa thủ tục cửa sổ của lớp window	
hIcon	Định danh của biểu tượng	Dùng hàm **LoadIcon**
hCursor	Định danh của con trỏ chuột	Dùng hàm **LoadCursor**
hbrBackground	Định danh của chổi tô nền	Dùng hàm **GetStockObject**
lpszMenuName	Tên thực đơn	Tên thực đơn gắn với cửa sổ, thực dơn này được khai báo trong tập tin tài nguyên.
lpszClassName	Tên lớp	

Bảng 1.2 Mô tả thuộc tính của lớp cửa sổ

Đăng ký lớp cửa sổ:

Sau khi định nghĩa một lớp cửa sổ, phải đăng ký lớp cửa sổ đó bằng hàm **RegisterClass**:

```
ATOM RegisterClass(CONST WNDCLASS * lpWndClass);
```

Tạo cửa sổ:

Lớp cửa sổ định nghĩa những đặc tính chung của cửa sổ, cho phép tạo ra nhiều cửa sổ dựa trên một lớp. Khi tạo ra một cửa sổ của hàm **CreateWindow**, ta chỉ định các đặc tính riêng của cửa sổ này, và phân biệt nó với các cửa sổ khác tạo ra cùng một lớp.

Khai báo hàm tạo cửa sổ:

```
HWND CreateWindow(LPCSTR lpClassName, // Tên lớp cửa sổ đã đăng ký
                  LPCSTR lpwindowName, // Tên của cửa sổ
                  DWORD dwStyle, // Kiểu của cửa sổ
                  int x, // Vị trí ngang ban đầu
                  int y, // Vị trí dọc ban đầu
                  int nWidth, // Độ rộng ban đầu
                  int nHeight, // Độ cao ban đầu
                  HWND hWndParent, // Định danh của cửa sổ cha
                  MENU hMenu, // Định dang của thực đơn
                  INSTANCE hInstance, // Định danh thể hiện ứng dụng
                  PVOID lpParam // Các tham số ban đầu
      );
```

Hiển thị cửa sổ:

Sau khi gọi hàm **CreateWindow**, một cửa sổ được tạo ra bên trong Windows, điều này có ý nghĩa là Windows đã cáp phát một vùng nhớ để lưu giữ tất cả các thông tin về cửa sổ đã được chỉ định trong hàm **CreateWindow**. Những thông số này sẽ được Windows tìm lại khi cần thiết dựa vào định danh mà hàm tạo cửa sổ trả về. Tuy nhiên, lúc này cửa sổ chưa xuất hiện trên màn hình Windows, để xuất hiện cần phải gọi hàm **ShowWindow**.

Hàm **ShowWindow** có khai báo như sau:

```
BOOL ShowWindow(HWND hWnd, // Định danh của cửa sổ cần thể hiện
                int nCmdShow // Trạng thái hiển thị
    );
```

Một số trạng thái của tham số nCmdShow:

- SW_HIDE: Ẩn cửa sổ.
- SW_MAXIMIZE: Phóng cửa sổ ra toàn bộ màn hình.
- SW_MINIMIZE: thu nhỏ thành biểu tượng trên màn hình.
- SW_RESTORE: Hiển thị dưới dạng chuẩn.

4. Hàm xử lý cửa sổ WndProc

Một chương trình Windows có thể chứa nhiều hơn một hàm xử lý cửa sổ. Một hàm xử lý cửa sổ luôn kết hợp với một lớp cửa sổ đặc thù. Hàm xử lý cửa sổ thường được đặt tên **WndProc**.

Hàm **WndProc** có chức năng giao tiếp với bên ngoài, tức là với Windows, toàn bộ các thông điệp gởi đến cửa sổ điều được xử lý qua hàm này.

Hàm này thường được khai báo như sau:

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);

Trong đó tham số đầu tiên là định danh của cửa sổ, tham số thứ 2 là định danh thông điệp, và cuối cùng là 2 tham số **WPARAM** và **LPARAM** bổ sung thông tin kèm theo thông điệp.

Chúng ta sẽ tìm hiểu một hàm xử lý cửa sổ **WndProc** sau:

```
LRESULT CALLBACK WndProc (HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    HDC hdc;
    PAINTSTRUCT ps;
    RECT rect;
    /*Xử lý các thông điệp cần thiết với ứng dụng*/
    switch (msg){
        case WM_CREATE:
            /*Viết đoạn mã khi tạo cửa sổ*/
            return 0;
        case WM_PAINT:
            /*Viết đoạn mã khi tô vẽ lại cửa sổ*/
            hdc = BeginPaint (hwnd, &ps);
            GetClientRect (hwnd, &rect);
            DrawText(hdc, "Hello", -1, &rect,
                     DT_SINGLELINE| DT_CENTER| DT_VCENTER);
```

```
            EndPaint(hwnd, &ps);
            return 0;
        case WM_SIZE:
            /*Viết đoạn mã khi kích thước cửa sổ thay đổi*/
            return 0;
        case WM_DESTROY:
             /*Cửa sổ bị đóng*/
             PostQuitMessage (0);
             return 0;
    }
    return DefWindowProc (hwnd, msg, wParam, lParam);
}
```

Thông thường chúng ta chỉ chặn để xử lý các thông điệp cần thiết có liên quan đến chức năng của ứng dụng. Các thông điệp khác thì giao cho hàm xử lý mặc định làm việc (hàm **DefWindowProc**).

5. Xử lý thông điệp

Sau khi cửa sổ được hiển thị trên màn hình, thì chương trình phải đọc các thông tin nhập của người dùng từ bàn phím hay thiết bị chuột. Windows sẽ duy trì một hàng đợi thông điệp cho mỗi chương trình chạy trên nó. Khi một sự kiện nhập thông tin xuất hiện, Windows sẽ dịch sự kiện này thành dạng thông điệp và đưa nó vào hàng đợi thông điệp của ứng dụng tương ứng.

Một ứng dụng nhận các thông điệp từ hàng đợi thông điệp bằng cách thực thi một đoạn mã sau:

```
while (GetMessage(&msg, NULL, 0,0)) {
    TranslateMessage (&msg);
    DispatchMessage (&msg);
}
```

Trong đó msg là một biến cấu trúc kiểu **MSG** được định nghĩa trong tập tin tiêu đề **WINUSER.H**.

```
typedef struct tagMSG {
    HWND hwnd;
    UINT message;
    WPARAM wParam;
    LPARAM lParam;
    DWORD time;
    POINT pt;
} MSG, *PMSG;
```

Kiểu dữ liệu **POINT** là một kiểu cấu trúc khác, được định nghĩa trong tập tin tiêu đề **WINDEF.H**, và có mô tả:

```
typedef struct tagPOINT {
    LONG x;
    LONG y;
} POINT, *PPOINT;
```

Ý nghĩa của các trường trong cấu trúc **MSG**

- ✓ hwnd: Định danh của cửa sổ mà thông điệp phát sinh.
- ✓ message: Định danh của thông điệp, ví dụ như thông điệp phát sinh khi bấm nút chuột trái là **WM_LBUTTONDOWN** có giá trị **0x0201**.
- ✓ wParam: Tham số 32-bit chứa các thông tin phụ thuộc vào từng thông điệp cụ thể.
- ✓ lParam: Tham số 32-bit phụ thuộc vào thông điệp.
- ✓ time: Thời gian đặt thông điệp trong hàng đợi.
- ✓ pt: Tọa độ của chuột khi đặt thông điệp vào hàng đợi

Hàm **GetMessage** sẽ trả về 0 nếu msg chứa thông điệp có định danh **WM_QUIT** (0x0012), khi đó vòng lặp thông điệp ngưng và ứng dụng kết thúc. Ngược lại thì hàm sẽ trả về một giá trị khác 0 với các thông điệp khác.

6. Xây dựng một ứng dụng đầu tiên

Một ứng dụng thường có giao diện nền tảng là một khung cửa sổ, để tạo được cửa sổ này chúng ta thực hiện bằng cách khai báo một lớp cửa sổ và đăng ký lớp cửa sổ đó. Để cửa sổ tương tác được thì chúng ta phải viết hàm xử lý cửa sổ **WndProc** khi đó tất cả các thông điệp liên quan đến cửa sổ sẽ được truyền vào

cho hàm này. Đoạn chương trình sau là khung sườn cho các chương trình viết trên Windows, bao gồm 2 hàm chính là:

WinMain: hàm chính của chương trình thực hiện các chức năng:

- Khai báo lớp cửa sổ.
- Đăng ký lớp cửa sổ vừa khai báo.
- Tạo và hiển thị lớp cửa sổ trên.
- Vòng lặp nhận thông điệp.

WndProc: Hàm xử lý thông điệp gởi đến cửa sổ.

```
/* HELLOWORLD.C */
#include <windows.h>
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine,
int iCmdShow)
{
    static TCHAR szAppName [] = TEXT ("HelloWorld");
    HWND hwnd;
    MSG msg;
    WNDCLASS wndclass;
    wndclass.style = CS_HREDRAW|CS_VREDRAW;
    wndclass.lpfnWndProc = WndProc;
    wndclass.cbClsExtra = 0;
    wndclass.cbWndExtra = 0;
    wndclass.hInstance = hInstance;
    wndclass.hIcon = LoadIcon (NULL, IDI_APPLICATION);
    wndclass.hCursor = LoadCursor (NULL, IDC_ARROW);
    wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH);
    wndclass.lpszMenuName = NULL;
    wndclass.lpszClassName = szAppName;
    if (!RegisterClass (&wndclass)) {
        MessageBox(NULL, TEXT (" The program requires Windows"), szAppName,
                   MB_ICONERROR);
        return 0;
    }
    hwnd = CreateWindow(szAppName, // Tên lớp cửa sổ
                        TEXT (" The Hello World Program"), // Tiêu đề cửa sổ
                        WS_OVERLAPPEDWINDOW, // Kiểu cửa sổ
                        CW_USEDEFAULT, // Toạ độ x
                        CW_USEDEFAULT, // Toạ độ y
                        CW_USEDEFAULT, // Chiều ngang
                        CW_USEDEFAULT, // Chiều dọc
                        NULL, // Cửa sổ cha
                        NULL, // Thực đơn
                        hInstance, // Định danh
                        NULL); // Tham số
    ShowWindow (hwnd, iCmdShow);
    UpdateWindow (hwnd);
    while (GetMessage (&msg, NULL, 0, 0)) {
          TranslateMessage (&msg);
          DispatchMessage (&msg);
    }
    return msg.wParam;
} // End WinMain

LRESULT CALLBACK WndProc (HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    HDC hdc;
    PAINTSTRUCT ps;
    RECT rect;
    switch (msg) {
```

```
        case WM_CREATE:
            return 0;
        case WM_PAINT:
            hdc = BeginPaint (hwnd, &ps);
            GetClientRect (hwnd, &rect);
            DrawText(hdc, TEXT("Hello World"), -1, &rect,
                     DT_SINGLELINE | DT_CENTER | DT_VCENTER);
            EndPaint (hwnd, &ps);
            return 0;
        case WM_DESTROY:
            PostQuitMessage (0);
            return 0;
    } // End switch
    return DefWindowProc (hwnd, msg, wParam, lParam);
}
```

Bảng dưới đây liệt kê ý nghĩa của các hàm được sử dụng trong 2 hàm **WinMain** và **WndProc** của chương trình HELLOWORLD.C.

Tên hàm	Ý nghĩa
LoadIcon	Nạp một biểu tượng để sử dụng trong chương trình.
LoadCursor	Nap một con trỏ chuột cho chương trình.
GetStockObject	Nhận một đối tượng đồ họa, trong trường hợp của chương trình thì lấy một chổi tô để tô lại nền của cửa sổ.
RegisterClass	Đăng ký một lớp cửa sổ cho cửa sổ ứng dụng trong chương trình.
MessageBox	Hiển thị một thông điệp.
CreateWindow	Tạo một cửa sổ dựa trên một lớp cửa sổ.
ShowWindow	Hiển thị cửa sổ lên màn hình.
UpdateWindow	Yêu cầu cửa sổ vẽ lại chính bản thân nó.
GetMesssage	Nhận một thông điệp từ hàng đợi thông điệp.
TranslateMessage	Dịch thông điệp bàn phím.
DispatchMessage	Gởi thông điệp đến hàm xứ lý cửa sổ.
BeginPaint	Khởi tạo chức năng vẽ của cửa sổ.
GetClientRect	Lấy hình chữ nhật lưu vùng làm việc.
DrawText	Hiển thị một chuỗi văn bản.
EndPaint	Kết thúc việc vẽ cửa sổ.
PostQuitMessage	Đưa thông điệp thoát vào hàng đợi thông điệp.
DefWindowProc	Thực hiện việc xử lý mặc định các thông điệp.

Bảng 1.3 Mô tả các hàm được sử dụng trong chương trình minh họa

7. Một số qui ước đặt tên biến

Khi viết một chương trình ứng dụng lớn với nhiều kiểu khai báo biến khác nhau, nếu việc khai báo các tên biến không thích hợp sẽ làm cho chương trình phức tạp thêm, đôi khi làm khó ngay cả người viết ra các mã nguồn đó. Vì vậy các lập trình viên thường qui ước sao cho một tên biến vừa gợi được chức năng của nó vừa xác định được kiểu loại. Có rất nhiều phong cách để đặt tên, trong số đó thì có phong cách đặt tên theo **cú pháp Hungary** (*Hungarian Notation*) là được dùng nhiều nhất. Qui ước rất đơn giản là bắt đầu tên biến thì viết chữ thường và các chữ đầu thể hiện kiểu dữ liệu của biến, và được gọi là các tiền tố. Ví dụ như biến **szCmdLine** là một biến lưu chuỗi nhập từ dòng lệnh, **sz** là thể hiện cho biến kiểu chuỗi kết thúc ký tự 0, ngoài ra ta hay thấy **hInstance** và **hPrevInstance**, trong đó **h** viết tắt cho kiểu handle, kiểu dữ liệu nguyên thường được khai báo dạng tiến tố là chữ i.

Cú pháp Hungary này giúp cho người lập trình rất nhiều trong khâu kiểm tra lỗi của chương trình, vì khi nhìn vào hai biến ta có thể dễ dàng nhận biết được sự không tương thích giữa hai kiểu dữ liệu thể hiện trong tên của hai biến.

Bảng mô tả một số tiền tố khi đặt tên biến của các kiểu dữ liệu:

Tiền tố	Kiểu dữ liệu
c	char, WCHAR, TCHAR
by	BYTE
n	short

i	int
x,y	biến lưu tọa độ x, y
b	BOOL
w	WORD
l	long
dw	DWORD
s	string
sz	chuỗi kết thúc bởi kí tự 0
h	handle
p	pointer
Lpsz	con trỏ dài chuỗi ký tự kết thúc kí tự 0

Bảng 1.4 Mô tả kiểu đặt tên biến

CHƯƠNG II: HỘP THOẠI VÀ THANH TRÌNH ĐƠN

A. MỞ ĐẦU

Hộp thoại (d*ialog*) và thanh trình đơn (*menu*) là các thành phần không thể thiếu trong việc tổ chức giao tiếp giữa người sử dụng và chương trình. Hộp thoại được xem như là một loại cửa sổ đặc biệt, là công cụ mềm dẻo, linh hoạt để đưa thông tin vào chương trình một cách dễ dàng. Trong khi menu là công cụ giúp người dùng thực hiện các thao tác đơn giản hơn, thông qua các nhóm chức năng thường sử dụng.

B. HỘP THOẠI

Hộp thoại phối hợp giữa người sử dụng với chương trình bằng một số phần tử điều khiển mà các phần tử này nhận nhiệm vụ thu nhận thông tin từ người dùng và cung cấp thông tin đến người dùng khi người dùng tác động đến các phần tử điều khiển. Các phần tử điều khiển này nhận cửa sổ cha là một hộp thoại. Các phần tử điều khiển thường là các **Button**, **List Box**, **Combo Box**, **Check Box**, **Radio Button**, **Edit Box**, **Scroll Bar**, **Static**.

Tương tự như các thông điệp gởi đến thủ tục **`WndProc`** của cửa sổ chính.Windows sẽ gởi các thông điệp xử lý hộp thoại đến thủ tục xử lý hộp thoại **`DlgProc`**. Hai thủ tục **`WndProc`** và thủ tục **`DlgProc`** tuy cách làm việc giống nhau nhưng giữa chúng có những điểm khác biệt cần lưu ý. Bên trong thủ tục xử lý hộp thoại bạn cần khởi tạo các phần tử điều khiển bên trong hộp thoại bằng thông điệp **`WM_INITDIALOG`**, cuối cùng là đóng hộp thoại, còn thủ tục xử lý **`WndProc`** thì không có. Có ba loại hộp thoại cơ bản. Hộp thoại trạng thái (*modal*), hộp thoại không trạng thái (*modeless*) và hộp thoại thông dụng (*common dialog*) mà chúng ta sẽ đề cập cụ thể trong các phần dưới.

1. Hộp thoại trạng thái

Hộp thoại trạng thái (modal) là loại hộp thoại thường dùng trong các ứng dụng của chúng ta. Khi hộp thoại trạng thái được hiển thị thì bạn không thể chuyển điều khiển đến các cửa sổ khác, điều này có nghĩa bạn phải đóng hộp thoại hiện hành trước khi muốn chuyển điều khiển đến các cửa sổ khác.

a. Cách tạo hộp thoại đơn giản

Sau đây là chương trình tạo ra một hộp thoại đơn giản. Hộp thoại được tạo ra có nội dung như sau.

Khi hộp thoại hiện lên có xuất hiện dòng chữ "`HELLO WORLD`", bên trên hộp thoại có một biểu tượng của hộp thoại đó là một **icon**, và phía dưới hộp thoại là một nút bấm (*Button*) có tên là **OK**, khi nhấp chuột vào nút **OK** thì hộp thoại "`HELLO WORLD`" được đóng lại.

Hình 2.1 Hộp thoại đơn giản

Đoạn code chương trình như sau (*Ví dụ 2.1*):

DIALOG.CPP *(trích dẫn)*

```
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK DialogProc (HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    static HINSTANCE hInstance;
    switch (message) {
        case WM_CREATE:
            hInstance = ((LPCREATESTRUCT) lParam)->hInstance;
```

```
            return 0;
        case WM_COMMAND:
            switch (LOWORD (wParam)){
                case IDC_SHOW:
                DialogBox (hInstance, TEXT ("DIALOG1"), hwnd, DialogProc);
                break;
            }
            return 0;
        case WM_DESTROY:
            PostQuitMessage (0);
            return 0;
    }
    return DefWindowProc (hwnd, message, wParam, lParam);
}

/*------------------hàm xử lý thông điệp hộp thoại------------------------------*/
BOOL CALLBACK DialogProc (HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
        case WM_INITDIALOG:
            return TRUE;
        case WM_COMMAND:
            switch (LOWORD(wParam)) {
                case IDOK:
                    EndDialog (hDlg, 0);
                    return TRUE;
            }
            Break ;
    }
    return FALSE;
}
```

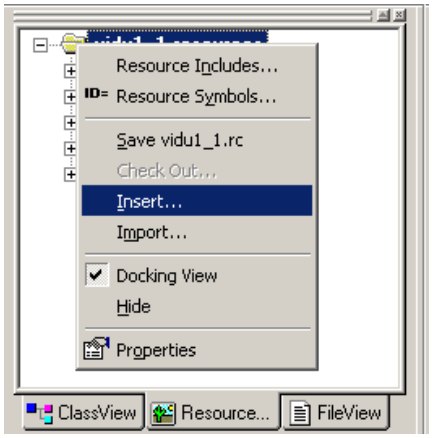
DIALOG1.RC *(trích dẫn)*

```
/*-----------------------------------dialog---------------------------------------*/
DIALOG1 DIALOG DISCARDABLE 40, 20, 164, 89
STYLE DS_MODALFRAME | WS_POPUP
FONT 9, "MS Sans Serif"
BEGIN
    DEFPUSHBUTTON "OK",IDOK,54,65,50,14
    CTEXT "HELLO WORLD ",IDC_STATIC,53,38,72,10
    ICON IDI_ICON1,IDC_STATIC,68,9,20,20
END
/* -----------------------------------Menu-----------------------------------------*/
MENU1 MENU DISCARDABLE
BEGIN
    POPUP "Dialog1"
    BEGIN
        MENUITEM "&Show", IDC_SHOW
    END
END
```

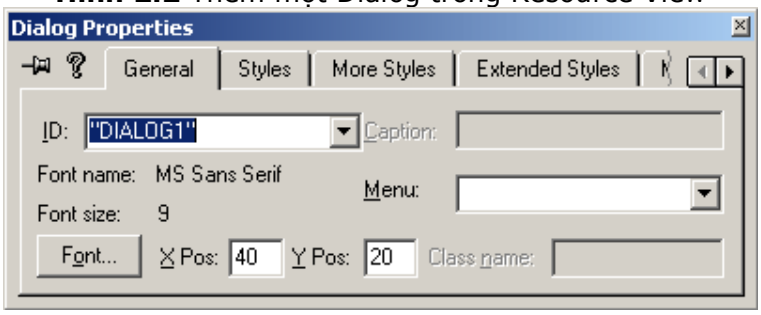
b. Hộp thoại và tạo mẫu template cho hộp thoại

Trong ví dụ 2.1 ở trên, ta đã tạo hộp thoại bằng cách dùng các câu lệnh chứa trong file tài nguyên DIALOG1.RC. Cách làm này giúp ta hiểu cấu trúc lệnh của Windows, tuy nhiên công cụ **Visual C++ Developer Studio**, ta có thể thiết lập một hộp thoại trực quan hơn như sau: Chọn **Insert** từ thực đơn **Resource View** để thêm một hộp thoại, màn hình được thể hiện như trong hình 2.2.

Miscrosoft sẽ hiển thị hộp thoại trực quan cùng với thanh công cụ để bạn có thể thêm các thành phần điều khiển vào hộp thoại. Chúng ta có thể điều chỉnh các thuộc tính của hộp thoại như tên hộp thoại, ID hộp thoại, ví trí hiển thị của hộp thoại trên cửa sổ chính, kích thước chữ và kiểu chữ thể hiện trên hộp thoại...vv bằng cách nhấn chuột phải trên hộp thoại thì cửa sổ **Properties** của hộp thoại được hiển thị (hình 2.3).



Hình 2.2 Thêm một Dialog trong Resource View



Hình 2.3 Hộp thoại Properties của Dialog

Trong cửa sổ **Properties** này chọn tab **Styles**, bỏ mục chọn **Title Bar** và không cần tạo tiêu đề cho cửa sổ. Sau đó đóng cửa sổ **Properties** của hộp thoại lại.

Bây giờ bắt đầu thiết kế diện mạo cho hộp thoại. Xóa nút **Cancel** vì không cần đến nút này. Để thêm một biểu tượng vào hộp thoại ta nhấn nút **Picture** lên thanh công cụ và kích chuột vào hộp thoại rồi kéo khung chữ nhật theo kích thước mong muốn. Đây là nơi mà biểu tượng được hiển thị. Nhấn chuột phải vào khung chữ nhật vừa tạo, chọn **Properties** từ trình đơn xuất hiện và để nguyên định danh của biểu tượng là IDC_STATIC. Định danh này sẽ được Windowns tự khai báo trong file Resource.h với giá trị -1. Giá trị -1 là giá trị của tất cả các định danh mà chương trình không cần tham chiếu đến. Tiếp đến là chọn đối tượng **Icon** trong trong mục **Type**, rồi gõ định danh của Icon cần thêm vào trong mục **Image**. Nếu đã tạo ra biểu tượng Icon trước thì chỉ việc chọn Icon từ danh sách các Icon trong mục Image.

Để thêm dòng chữ "HELLO WORLD" vào hộp thoại, chọn **Static Text** từ bảng công cụ và đặt đối tượng vào hộp thoại. Nhấn chuột phải để hiện thị **Properties** của Static Text, sau đó vào mục caption đánh dòng chữ "HELLO WORD" vào đây.

Dịch và chạy chương trình sau đó xem file DIALOG1.RC dưới dạng text, nội dung hộp thoại được Windows phát sinh như sau:

```
DIALOG1 DIALOG DISCARDABLE 40, 20, 164, 90
STYLE DS_MODALFRAME | WS_POPUP
FONT 9, "MS Sans Serif"
BEGIN
    DEFPUSHBUTTON "OK",IDOK,54,65,50,14
    CTEXT "HELLO WORLD ",IDC_STATIC,53,38,72,10
    ICON IDI_ICON1,IDC_STATIC,68,9,21,20
END
```

Dòng đầu tiên là tên của hộp thoại "DIALOG1" kế tiếp là từ khóa **DIALOG**, **DISCARDABLE** và tiếp sau đó là 4 số nguyên. Hai số nguyên đầu tiên chỉ vị trí dòng, cột của hộp thoại sẽ được hiển thị trên cửa sổ chính. Hai số nguyên tiếp theo xác định kích thước của hộp thoại theo thứ tự cột và dòng.

Lưu ý: Các thông số định tọa độ và kích thước của hộp thoại không tính theo đơn vị **Pixel** mà tính theo kích cở của **Font** chữ. Số đo của tọa độ x và chiều rộng dựa trên 1/4 đơn vị rộng trung bình của Font chữ. Số đo của tọa độ y và chiều cao dựa trên 1/8 đơn vị cao trung bình của Font chữ.

Theo sau lệnh **STYLE** là các thuộc tính của hộp thoại mà bạn cần thêm vào. Thông thường hộp thoại modal sử dụng các hằng **WS_POPUP** và **DS_MODALFRAME** ngoài ra còn có các hằng **WS_CAPTION, WS_MAXIMIZEBOX**, **WS_MINIMIZEBOX**, **WS_POPUP, WS_VSCROLL**, **WS_HSCROLL**, **WS_SYSMENU**, … Lệnh **BEGIN** và lệnh **END** có thể được thay bằng **{** và **}**. Trong ví dụ trên, hộp thoại sử dụng 3 kiểu điều

khiển là **DEFPUSHBUTTON** (kiểu nút bấm mặc định), **ICON** (biểu tượng), và kiểu **CTEXT** (văn bản được canh giữa). Một kiểu điều khiển được khai báo tổng quát như sau.

Control-type "text", id, xPos, yPos, xWidth, yHeight, iStyle.

Control-type là các từ khóa khai báo kiểu điều khiển như **DEFPUSHBUTTON**, **ICON**, **CTEXT**, …, **id** là định danh của các điều khiển, thông thường một điều khiển có một định danh riêng được gởi cùng với thông điệp **WM_COMMAND** đến các thủ tục xử lý thông điệp của cửa sổ cha. **xPos**, **yPos** là vị trí cột, dòng hiển thị của điều khiển đó trên cửa sổ cha. **xWidth**, **yHeight** là chiều rộng và chiều cao của điều khiển đó. Đối số cuối cùng là **iStyle**, đối số này tùy chọn dùng để định nghĩa thêm các kiểu cửa sổ mà điều khiển cần thể hiện chúng thường là các hằng **WS_** được khai báo trong tập tin "**.h**" của Windows.

c. Thủ tục xử lý thông điệp của hộp thoại

Thủ tục xử lý thông điệp của hộp thoại dùng để xử lý tất cả các thông điệp từ bộ quản lý hộp thoại của Windows gởi đến hôp thoại. Thủ tục này được Windows gọi khi có sự tác động lên các phần tử điểu khiển nằm trong hộp thoại.

Xét thủ tục xử lý hộp thoại **DialogProc** trong ví dụ 2.1. Thủ tục này có 4 tham số như thủ tục **WndProc**, và thủ tục này được định nghĩa kiểu trả về là **CALLBACK**.Tuy hai thủ tục này tương tự giống nhau nhưng thực sự giữa chúng có một vài sự khác biệt đáng chú ý.

- Thủ tục **DialogProc** trả về giá trị kiểu **BOOL,** trong khi thủ tục **WindProc** thì trả về giá trị **LRESULT**.
- Thủ tục **DialogProc** trả về giá trị **TRUE** (giá trị khác 0) nếu nó xử lý thông điệp và ngược lại nếu không xử lý các thông điệp thì thủ tục trả về giá thị là **FALSE** (trị 0). Còn thủ tục **WindProc** thì gọi hàm **DefWindowProc** với các thông điệp không cần xử lý.
- Thủ tục **DialogProc** không cần xử lý thông điệp **WM_DESTROY,** cũng không cần xử lý thông điệp **WM_PAINT** và cũng không nhận được thông điệp **WM_CREATE** mà là thông điệp **WM_INITDIALOG** dùng để khởi tạo hộp thoại.

Ngoài xử lý thông điệp **WM_INITDIALOG**, thủ tục xử lý thông điệp hộp thoại chỉ xử lý một thông điệp duy nhất khác là **WM_COMMAND**. Đây cũng là thông điệp được gởi đến cửa sổ cha khi ta kích hoạt (nút nhấn đang nhận được focus) lên các thành phần điểu khiển. Chỉ danh ID của nút "**OK**" là **IDOK** sẽ được chứa trong word thấp của đối số **wParam**. Khi nút này được nhấn, thủ tục **DialogProc** gọi hàm **EndDialog** để kết thúc xử lý và đóng hộp thoại.

Các thông điệp gửi đến hộp thoại không đi qua hàng đợi mà nó được Windows gọi trực tiếp hàm **DialogProc** để truyền các thông điệp vào cho thủ tục xử lý hộp thoại.Vì vậy, không phải bận tâm về hiệu ứng của các phím tắt được quy định trong chương trình chính.

d. Gọi hiển thị hộp thoại và các vấn đề liên quan

Trong thủ tục **WndProc** khi xử lý thông điệp **WM_CREATE** Windows lấy về định danh hInstance của chương trình và lưu nó trong biến tĩnh hInstance như sau:

```
hInstance = ((LPCREATESTRUCT) lParam)->hInstance;
```

Dialog1 kiểm tra thông điệp **WM_COMMAND** xem word thấp của đối số wParam có bằng giá trị **IDC_SHOW** (chỉ danh của thành phần **Show** trong thực đơn). Nếu phải, tức đã chọn mục **Show** trên trình đơn của cửa sổ chính và yêu cầu hiển thị hộp thoại, lúc này chương trình gọi hiển thị hộp thoại bằng cách gọi hàm.

```
DialogBox (hInstance, TEXT ("DIALOG1"), hwnd, DialogProc)
```

Đối số đầu tiên của hàm này phải là hInstance của chương trình gọi, đối số thứ hai là tên của hộp thoại cần hiển thị, đối số thứ 3 là cửa sổ cha mà hộp thoại thuộc về, cuối cùng là địa chỉ của thủ tục xử lý các thông điệp của hộp thoại.

Chương trình không thể trả điều khiển về hàm **WndProc** cho đến khi hộp thoại được đóng lại. Giá trị trả về của hàm **DialogBox** là giá trị của đối số thứ hai trong hàm **EndDialog** nằm bên trong thủ tục xử lý thông điệp hộp thoại. Tuy nhiên chúng ta cũng có thể gởi thông điệp đến hàm **WndProc** yêu cầu xử lý ngay cả khi hộp thoại đang mở nhờ hàm **SendMessage** như sau:

```
SendMessage(GetParent(hDlg), message, wParam, lParam)
```

Tuy Visual C++ Developer đã cung cấp cho chúng ta bộ soạn thảo hộp thoại trực quan mà ta không cần phải quan tâm đến nội dung trong tập tin.**RC**. Tuy nhiên với cách thiết kế một hộp thoại bằng các câu lệnh giúp chúng ta hiểu chi tiết hơn cấu trúc lệnh của Windows hơn thế nữa tập lệnh dùng để thiết kế hộp thoại phong phú và đa dạng hơn rất nhiều so với những gì mà ta trực quan được trên bộ soạn thảo của Developer. Bằng cách sử dụng các lệnh đặc biệt trong tập tin Resource editor của Visual C++ ta có thể tạo ra nhiều đối tượng mà trong bộ soạn thảo không có.

Thêm hằng **WS_THINKFRAME** vào mục **STYLE** để co giản hộp thoại (tương đương với trong boder ta chọn mục **Resizing**).

Để đặt nội dung tiêu đề cho hộp thoại ta chỉ việc thêm hằng **WS_CAPTION** trong **STYLE**.

```
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION
CAPTION "Hello Dialog1"
```

Có thể dùng cách khác để thêm tiêu đề cho hộp thoại, bằng cách trong khi xử lý thông điệp **WM_INITDIALOG** thêm vào dòng lệnh:

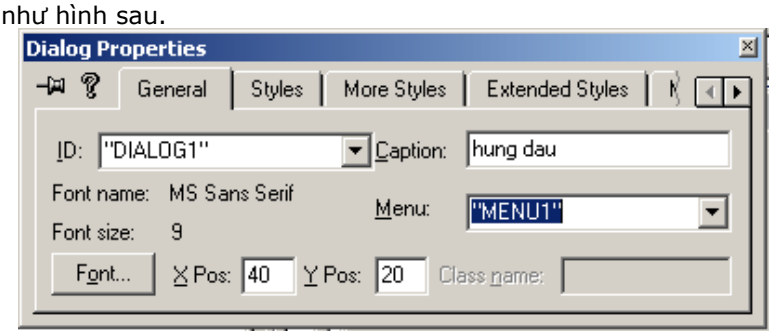
```
SetWindowText(hDlg,TEXT("Hello Dialog"));
```

Khi hộp thoại có tiêu đề rồi, có thể thêm các chức năng phóng to và thu nhỏ hộp thoại bằng hằng **WS_MINIMIZEBOX**, **WS_MAXIMIZEBOX**.

Có thể thêm trình đơn vào hộp thoại nếu muốn bằng đoạn lệnh.

```
DIALOG1 DIALOG DISCARDABLE 40, 20, 164, 90
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION
CAPTION "Hello Dialog1"
MENU MENU1
```

Trong đó **MENU1** là tên của trình đơn ta đã tạo. Trong Visual C++ Developer ta chỉ cần chọn tên thực đơn trong mục Menu như hình sau.



Hình 2.4 Chọn menu trong Dialog Propertier

Từ cửa sổ **Properties** trên thể chọn mục **"Font"** để định Font chữ cho hộp thoại.

Gọi hàm **DialogBoxIndirect** để tạo ra một hộp thoại mà không cần dùng resource script. Hộp thoại tạo ra bằng hàm này trong khi chương trình đang thực hiện được gọi là hộp thoại tạo tự động.

Trong ví dụ 3-1 ta chỉ dùng 3 kiểu điều khiển đó là các kiểu **'ICON'**, **'CTEXT'**, **'DEFPUSHBUTTON'**. Ngoài ra còn có các kiểu điều khiển được liệt kê trong bảng sau.

Kiểu điều khiển	Lớp cửa sổ	Kiểu cửa sổ
PUSHBUTTON	Button	BS_BUSHBUTTON
DEFPUSHBUTTON	Button	BS_DEFBUSHBUTTON | WS_TABSTOP
CHECKBOX	Button	BS_CHECKBOX | WS_TABSTOP
RADIOBUTTON	Button	BS_RADIOBUTTON | WS_TABSTOP
GROUPBOX	Button	BS_GROUPBOX | WS_TABSTOP
LTEXT	Static	SS_LEFT | WS_GROUP
CTEXT	Static	SS_CENTER | WS_GROUP
RTEXT	Static	SS_RIGHT | WS_GROUP
ICON	Static	SS_ICON
EDITTEXT	Edit	ES_LEFT | WS_BORDER | WS_STABSTOP
SCROLLBAR	Scrollbar	SBS_HORZ
LISTBOX	Listbox	LBS_NOTIFY | WS_BORDER | WS_VSCROLL
COMBOBOX	Combobox	CBS_SIMPLE | WS_TABSTOP

Bảng 2.1 Các kiểu điều khiển

Các kiểu điều khiển được khai báo trong resource script có dạng như sau, ngoại trừ kiểu điều khiển **LISTBOX**, **COMBOBOX**, **SCROLLBAR**, **EDITTEXT**.

```
Control-type "text", id, xPos, yPos, xWidth, yHeight, iStyle
```

Các kiểu điều khiển **LISTBOX**, **COMBOBOX**, **SCROLLBAR**, **EDITTEXT** được khai báo trong **resource script** với cấu trúc như trên nhưng không có trường **"text"**.

Thêm thuộc tính cho các kiểu điều khiển bằng cách thay đổi tham số iStyle. Ví dụ ta muốn tạo **radio button** với chuỗi diễn đạt nằm ở bên trái của nút thì ta gán trường iStyle bằng **BS_LEFTTEXT** cụ thể như sau.

RADIOBUTTON "Radio1",IDC_RADIO1,106,10,53,15,BS_LEFTTEXT

Trong **resource script** ta cũng có thể tạo một kiểu điều khiển bằng lệnh tổng quát sau.

CONTROL "text", id, "class", iStyle, xPos, yPos, xWidth, yHeight

Trong đó **class** là tên lớp muốn tạo ví dụ thay vì tạo một **radio button** bằng câu lệnh.

RADIOBUTTON "Radio1",IDC_RADIO1,106,10,53,15,BS_LEFTTEXT

Thay bằng đoạn lệnh sau:

CONTROL"Radio1",IDC_RADIO1,"button",106,10,53,15,BS_LEFTTEXT

e. Ví dụ chương trình về hộp thoại

Để minh họa cho việc trao đổi thông điệp giữa các thành phần điều khiển bên trong hộp thoại (đóng vai trò là một cửa sổ cha) với các thành phần điều khiển con nằm bên trong hộp thoại, và cơ chế quản lý hộp thoại của Windows. Chúng ta tiến hành xem xét ví dụ 2-2. Kết quả thực hiện của chương trình như trong hình 2.5.

Cửa sổ hộp thoại gồm có ba nhóm nút chọn radio.Nhóm thứ nhất dùng để chọn đối tượng vẽ là hình chữ nhật hay hình ellipse, nhóm thứ hai dùng để chọn màu tô cho hình vẽ, nhóm thứ 3 dùng để chọn kiểu tô cho hình vẽ. Khi thay đổi việc chọn màu tô, kiểu tô thì màu tô và kiểu tô của hình vẽ cạnh bên sẽ thay đổi theo màu tô, và kiểu tô vừa mới chọn. Khi nhấn nút **OK** thì hộp thoại đóng lại và màu tô, kiểu tô cùng hình vẽ vừa mới vẽ sẽ được hiển thị lên cửa sổ chính. Nếu nhấn nút **Cancel** hoặc nhấn phím **Esc** thì hộp thoại được đóng lại nhưng hình vẽ, màu tô và kiểu tô không được hiển thị lên cửa sổ chính. Trong ví dụ này nút **OK** và nút **Cancel** có chỉ danh ID lần lượt là **IDOK** và **IDCANCEL**.Thông thường đặt chỉ danh cho các phần tử điều khiển nằm trong hộp thoại được bắt đầu bằng chữ ID. Biểu tượng chiếc xe đạp trên hộp thoại đó là một icon. Trên thanh tiêu đề của cửa sổ chính có một biểu tượng, biểu tượng đó cũng là một **icon** (đó là một ly trà). Khi đặt các nút **radio** vào hộp thoại bằng công cụ Developer studio nhớ phải đặt các nút đó theo thứ tự như hình 2-5. Thì khi đó Windows mới phát sinh mã cho các nút đó theo thứ tự tăng dần, điều này giúp chúng ta dễ dàng kiểm soát các thao tác trên tập các nút radio. Bạn nhớ bỏ luôn mục chọn **Auto** trong phần thiết lập **Properties** của các nút chọn radio. Bởi vì các nút radio mang thuộc tính **Auto** yêu cầu viết ít mã lệnh hơn nhưng chúng thường khó hiểu so với các nút không có thuộc tính **Auto**. Chọn thuộc tính **Group**, **Tab stop** trong phần thiết kế **Properties** của nút **OK**, nút **Cancel**, và hai nút radio đầu tiên trong ba nhóm radio để có thể chuyển focus (*chọn*) bằng phím **Tab** trên bàn phím.

Hình 2.5 Minh họa trao đổi thông điệp qua các điều khiển

Chương trình minh họa (*Ví dụ 2.2*):

DIALOG2.CPP *(trích dẫn)*

```
#include <windows.h>
#include "resource.h"
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
BOOL CALLBACK DialogProc (HWND, UINT, WPARAM, LPARAM);
int iCurrentColor = IDC_BLACK, iCurrentFigure = IDC_RECT;
int iCurrenBrush = IDC_HS_BDIAGONAL;
void PaintWindow(HWND hwnd, int iColor, int iFigure, int iBrush) {
    static COLORREF crColor[8] = {RGB(0, 0, 0), RGB(0, 0, 255), RGB(0, 255, 0),
                                  RGB(0, 255, 255), RGB(255, 0, 0),
                                  RGB(255, 0, 255), RGB(255, 255, 0),
                                  RGB(255, 255, 255)};
    HBRUSH hBrush, hbrush;
    HDC hdc;
    RECT rect;
    hdc = GetDC(hwnd);
    GetClientRect(hwnd, &rect);
```

ID của Radio BLack

```
    if (iBrush == IDC_HS_BDIAGONAL)
        hbrush = CreateHatchBrush(HS_BDIAGONAL, crColor[iColor-IDC_BLACK]);
    if (iBrush == IDC_HS_CROSS)
        hbrush = CreateHatchBrush(HS_CROSS, crColor[iColor - IDC_BLACK]);
    if (iBrush == IDC_HS_DIAGCROSS)
        hbrush = CreateHatchBrush(HS_DIAGCROSS, crColor[iColor - IDC_BLACK]);
    if (iBrush == IDC_HS_FDIAGONAL)
        hbrush = CreateHatchBrush(HS_FDIAGONAL, crColor[iColor - IDC_BLACK]);
    if (iBrush == IDC_HS_HORIZONTAL)
        hbrush = CreateHatchBrush(HS_HORIZONTAL, crColor[iColor - IDC_BLACK]);
    if (iBrush == IDC_HS_VERTICAL)
        hbrush = CreateHatchBrush(HS_BDIAGONAL, crColor[iColor - IDC_BLACK]);
    hBrush = (HBRUSH) SelectObject (hdc, hbrush);
    if (iFigure == IDC_RECT)
        Rectangle (hdc, rect.left, rect.top, rect.right, rect.bottom);
    else
        Ellipse(hdc, rect.left, rect.top, rect.right, rect.bottom);
    DeleteObject (SelectObject (hdc, hBrush));
    ReleaseDC (hwnd, hdc);
}

void PaintTheBlock(HWND hCtrl, int iColor, int iFigure, int iBrush) {
    InvalidateRect (hCtrl, NULL, TRUE);
    UpdateWindow (hCtrl);
    PaintWindow (hCtrl, iColor, iFigure,iBrush);
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) {
    static HINSTANCE hInstance;
    PAINTSTRUCT ps;
    switch (message) {
        case WM_CREATE:
            hInstance = ((LPCREATESTRUCT) lParam)->hInstance;
            return 0;
        case WM_COMMAND:
            switch (LOWORD (wParam)) {
                case IDC_SHOW:
                    if (DialogBox (hInstance, TEXT ("DIALOG"), hwnd, DialogProc))
                    InvalidateRect (hwnd, NULL, TRUE);
                    return 0;
            }
            break;
        case WM_PAINT:
            BeginPaint (hwnd, &ps);
            EndPaint (hwnd, &ps);
            PaintWindow (hwnd, iCurrentColor, iCurrentFigure, iCurrenBrush);
            return 0;
        case WM_DESTROY:
            PostQuitMessage (0);
            return 0;
    }
    return DefWindowProc (hwnd, message, wParam, lParam);
}

BOOL CALLBACK DialogProc(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam) {
    static HWND hCtrlBlock;
    static int iColor, iFigure, iBrush;
    switch (message) {
        case WM_INITDIALOG:
```

```
            iColor = iCurrentColor;
            iFigure = iCurrentFigure;
            iBrush = iCurrenBrush;
            CheckRadioButton(hDlg,IDC_BLACK,IDC_WHITE, iColor);
            CheckRadioButton(hDlg,IDC_RECT,IDC_ELLIPSE, iFigure);
            CheckRadioButton(hDlg, IDC_HS_BDIAGONAL, IDC_HS_VERTICAL, iBrush);
            hCtrlBlock = GetDlgItem (hDlg, IDC_PAINT);
            SetFocus (GetDlgItem (hDlg, iColor));
            return FALSE;
        case WM_COMMAND:
            switch (LOWORD (wParam)) {
                case IDOK:
                    iCurrentColor = iColor;
                    iCurrentFigure = iFigure;
                    iCurrenBrush = iBrush;
                    EndDialog (hDlg, TRUE);
                    return TRUE;
                case IDCANCEL:
                    EndDialog (hDlg, FALSE);
                    return TRUE;
                case IDC_BLACK:
                case IDC_RED:
                case IDC_GREEN:
                case IDC_YELLOW:
                case IDC_BLUE:
                case IDC_MAGENTA:
                case IDC_CYAN:
                case IDC_WHITE:
                    iColor = LOWORD (wParam);
                    CheckRadioButton (hDlg, IDC_BLACK, IDC_WHITE, LOWORD(wParam));
                    PaintTheBlock (hCtrlBlock, iColor, iFigure, iBrush);
                    return TRUE;
                case IDC_RECT:
                case IDC_ELLIPSE:
                    iFigure = LOWORD(wParam);
                    CheckRadioButton(hDlg, IDC_RECT, IDC_ELLIPSE, LOWORD(wParam));
                    PaintTheBlock (hCtrlBlock, iColor, iFigure, iBrush);
                    return TRUE;
                case IDC_HS_BDIAGONAL:
                case IDC_HS_CROSS:
                case IDC_HS_DIAGCROSS:
                case IDC_HS_FDIAGONAL:
                case IDC_HS_HORIZONTAL:
                case IDC_HS_VERTICAL:
                    iBrush = LOWORD(wParam);
                    CheckRadioButton(hDlg, IDC_HS_BDIAGONAL, IDC_HS_VERTICAL,
                                     LOWORD(wParam));
                    PaintTheBlock(hCtrlBlock, iColor, iFigure, iBrush);
                    return TRUE;
            }
            break;
        case WM_PAINT:
            PaintTheBlock (hCtrlBlock, iColor, iFigure,iBrush);
            break;
    }
    return FALSE;
}
```

f. Làm việc với các thành phần điều khiển trong hộp thoại

Các thành phần điều khiển con đều gởi thông điệp **WM_COMMAND** đến cửa sổ cha của nó và cửa sổ cha có thể thay đổi trạng thái của các thành phần điều khiển con như kích hoạt, đánh dấu (*check*), bỏ dấu check (*uncheck*) bằng cách gởi các thông điệp đến các thành phần điều khiển con nằm trong nó. Tuy nhiên trong Windows đã cung cấp cơ chế trao đổi thông điệp giữa các thành phần điều khiển con với cửa sổ cha. Chúng ta bắt đầu tìm hiểu các cơ chế trao đổi thông điệp đó.

Trong ví dụ 2.2 mẫu template của hộp thoại Dialog2 được thể hiện trong tập tin tài nguyên DIALOG2.RC gồm có các thành phần. Thành phần **GROUPBOX** có tiêu đề do chúng ta gõ vào, thành phần này chỉ đơn giản là một khung viền bao quanh hai nhóm nút chọn radio, và hai nhóm này hoàn toàn độc lập với nhau trong mỗi nhóm.

Khi một trong những nút radio được kích hoạt thì cửa sổ điều khiển con gởi thông điệp **WM_COMMAND** đến cửa sổ cha (ở đây là hộp thoại) với word thấp của đối số wParam chứa thành phần ID của điều khiển con, word cao của đối số wParam cho biết mã thông báo. Sau cùng là đối số lParam mang handle của cửa sổ điều khiển con. Mã thông báo của nút chọn radio luôn luôn là **BN_CLICKED** (mang giá trị 0). Windows sẽ chuyển thông điệp **WM_COMMAND** cùng với các đối số wParam và lParam đến thủ tục xử lý thông điệp của hộp thoại (**DialogProc**). Khi hộp thoại nhận được thông điệp **WM_COMMAND** cùng với các đối số lParam và wParam, hộp thoại kiểm tra trạng thái của tất cả các thành phần điều khiển con nằm trong nó và thiết lập các trạng thái cho các thành phần điều khiển con này.

Có thể đánh dấu một nút chọn bằng cách gởi thông điệp

```
SendMessage (hwndCtrl,MB_SETCHECK, 1, 0);
```

Và ngược lại muốn bỏ chọn một nút nào đó thì dùng hàm.

```
SendMessage (hwndCtrl, MB_SETCHECK, 0, 0);
```

Trong đó đối số hwndCtrl là handle của cửa sổ điều khiển con.

Chúng ta có thể gặp rắc rối khi muốn sử dụng hai hàm trên bởi vì không biết handle của các thành phần điều khiển con. Chúng ta chỉ biết handle của các thành phần điều khiển con khi nhận được thông điệp **WM_COMMAND**. Để giải quyết được vướng mắc trên, trong Windows cung cấp một hàm để lấy handle của cửa sổ con khi biết được định danh ID của nó bằng hàm.

```
hwndCtrl = GetDlgItem (hDlg, id); // hDlg là handle của hộp thoại
```

Có thể lấy được chỉ danh ID của thành phần điều khiển con khi biết được handle của nó bằng hàm sau.

```
id = GetWindowLong (hwndCtrl, GWL_ID);
```

Tuy nhiên, chúng ta có thể quản lý ID của các thành phần điều khiển con, còn handle là do Windows cấp ngẫu nhiên, do đó việc dùng handle để nhận về ID của các thành phần điều khiển con là ít dùng đến.

Khi hộp thoại nhận được thông điệp **WM_COMAND** thì chúng ta phải kiểm tra nút radio nào được chọn (xác định màu cần chọn), và tiến hành bỏ chọn các nút khác bằng đoạn lệnh sau.

```
case WM_COMMAND:
    switch (LOWORD(wParam)) {
        case IDC_BLACK:
        case IDC_RED:
        case IDC_GREEN:
        case IDC_YELLOW:
        case IDC_BLUE:
        case IDC_MAGENTA:
        case IDC_CYAN:
        case IDC_WHITE:
            iColor = LOWORD (wParam);
            for (i = IDC_BLACK, i < IDC_WHITE,i++)
                SendMessage(GetDlgItem(hDlg, i), MB_SETCHECK, i == LOWORD(wParam),
0).
            return TRUE;
    }
```

Trong đó iColor dùng để lưu giá trị màu hiện hành được chọn. Vòng lặp for dùng để kiểm tra trạng thái của tất cả các nút radio thông qua ID của chúng. Hàm **GetDlgItem** dùng để lấy handle của nút được chọn và lưu vào biến i. Hàm **SendMessage** dùng để gởi thông điệp **MB_SETCHECK** tới các nút radio. Nếu word thấp của đối số wParam bằng chỉ danh ID của nút được chọn thì nút đó được đánh dấu và các nút khác sẽ không được chọn.

*Chú ý:*Trong các ví dụ trên thường dùng hai nút **OK** và nút **Cancel**, hai nút này được Windows đặt định danh mặc định theo thứ tự là **IDOK** và **IDCANCEL**. Thông thường đóng hộp thoại bằng cách nhấn chuột vào một trong hai nút **OK** hoặc **Cancel**. Trong Windows, khi nhấn nút Enter thì Windows luôn phát sinh thông điệp **WM_COMMAND**, bất kỳ đối tượng nào đang nhận focus. **LOWORD** của đối số wParam mang giá trị

ID của nút nhấn mặc định (nút OK), ngoài trừ có một nút đang nhận focus (trong trường hợp này thì **LOWORD** của đối số wParam mang chỉ danh của nút đang nhận focus). Nếu nhấn nút **Esc** hay nhấn **Ctrl+Break**, thì Windows gởi thông điệp **WM_COMMAND** với thành phần **LOWORD** của đối số wParam có giá trị **IDCANCEL** (định danh mặc định của nút **Cancel**). Do đó không cần phải xử lý thêm các phím gõ để đóng hộp thoại.

Trong ví dụ 2.2 để xử lý hai trường hợp khi nhấn nút **Cancel** và nút **OK** ta dùng đoạn chương trình sau.

```
switch (LOWORD (wParam))
{
    case IDOK:
        iCurrentColor = iColor;
        iCurrentFigure = iFigure;
        EndDialog (hDlg, TRUE);
        return TRUE;
    case IDCANCEL:
        EndDialog (hDlg, FALSE);
        return TRUE;
        …
}
```

Hàm **EndDialog** dùng để kết thúc và đóng hộp thoại.Trong trường hợp nhấn nút **OK** thì hai giá trị iCurrentColor và giá trị iCurrentFigure được lưu lại cho cửa sổ cha (cả hai biến trên đều là biến toàn cục). Chú ý rằng, hai giá trị khác biệt (TRUE, FALSE) của đối số thứ hai trong lời gọi hàm **EndDialog**. Giá trị này sẽ được trả ngược về từ lời gọi hàm **DialogBox** trong thủ tục **WndProc**.

```
case WM_COMMAND:
    switch (LOWORD(wParam)) {
        case IDC_SHOW:
            if(DialogBox(hInstance, TEXT("DIALOG"), hwnd, DialogProc))
            InvalidateRect (hwnd, NULL, TRUE);
            return 0;
    }
    break;
```

Có nghĩa nếu hàm **DialogBox** trả về giá trị TRUE, tức nút **OK** được nhấn. Lúc đó thủ tục **WndProc** sẽ cập nhật lại nội dung của cửa sổ chính, bằng cách ghi lại sự thay đổi giá trị của hai biến toàn cục `iCurrentColor` và giá trị `iCurrentFigure` dùng để vẽ lại hình chữ nhật hay hình ellipse với màu được chọn là `iCurrentColor`.

Và ngược lại nếu nhấn nút **Cancel** thì giá trị `iCurrentColor` và giá trị `iCurrentFigure` sẽ không thay đổi, tức thủ tục **WndProc** sử dụng lại giá trị cũ.

Giá trị `TRUE` hay `FALSE` thông báo cho cửa sổ chính biết rằng người dùng từ chối hay chấp thuận tùy chọn trong hộp thoại. Vì `TRUE` và `FALSE` có kiểu số nguyên (1,0) nên đối số thứ hai trong lời gọi hàm **EndDialog** có kiểu số nguyên (**int**). Do đó kết quả trả về của hàm này cũng có kiểu là số nguyên. Ví dụ nếu bạn bấm nút **OK** thì trị trả về của hàm bằng 1. Nếu bạn bấm nút **Cancel** thì trị trả về của hàm bằng 0, và nếu trong chương trình có sử dụng nút bấm mặc định **Ignore** thì khi bấm nút này trị trả của hàm sẽ là 2.

g. Vẽ trong hộp thoại

Trong ví dụ 2.2 chúng ta đã dùng phương pháp vẽ trên hộp thoại đây là công việc khác thường. Bây giờ ta tìm hiểu công việc đó tiến hành như thế nào.

Trong file RESOURCE.RC có thành phần điều khiển là.

LTEXT "",IDC_PAINT, 5, 22, 92, 93

Khi chúng ta chọn nút radio để thay đổi màu, hình vẽ hay nhận được thông điệp **WM_PAINT** thì thủ tục **DialogProc** thực hiện thao tác vẽ vào thành phần điều khiển của hộp thoại bằng hàm **PaintTheBlock**. Hàm này được khai báo như sau.

PaintTheBlock(hCtrBlock, iColor, iFigure);

Trong đó hCtrBlock là handle của thành phần điều khiển có định danh là **IDC_PAINT**. Handle của thành phần điều khiển này được lấy về bởi hàm.

hCtrBlock=**GetDlgItem**(hDlg, IDC_PAINT);

Nội dung của hàm **PaintTheBlock** như sau.

```
void PaintTheBlock(HWND hCtrl, int iColor, int iFigure)
{
```

```
    InvalidateRect(hCtrl, NULL, TRUE);
    UpdateWinDow(hCtrl);
    PaintWinDow(hCtrl, iColor, iFigure);
}
```

Lệnh **InvalidateRect(**hCtrl, **NULL, TRUE)** và **UpdateWindow(**hCtrl**)** có nhiệm vụ làm cho cửa sổ con cần phải vẽ lại. Hàm PaintWindow dùng để vẽ ra màn hình ellipse hay chữ nhật. Đầu tiên hàm này lấy **DC** (device context) của thiết bị có handle là hCtrl, và vẽ lên thiết bị này dạng hình ảnh cùng với màu tô được chọn. Kích thước của cửa sổ con cần vẽ được lấy bằng hàm **GetClientRect**. Hàm này trả về kích thước của vùng client cần vẽ theo đơn vị tính là pixel.

Chúng ta vẽ trên vùng client của các điều khiển con chứ không vẽ trực tiếp lên vùng client của hộp thoại. Khi hộp thoại nhận được thông điệp **WM_PAINT** thì thành phần điều khiển có định danh **IDC_PAINT** được vẽ lại. Cách xử lý thông điệp **WM_PAINT** giống như thủ tục xử lý **WndProc** của cửa sổ chính, nhưng thủ tục xử lý hộp thoại không gọi hàm **BeginPaint** và hàm **EndPaint** bởi vì nó không tự vẽ lên cửa sổ của chính nó.

Nếu muốn vô hiệu hóa một phần tử điều kiển, tức biến đổi nút sang trạng thái vô hiệu hóa thì dùng hàm.

EnableWindow(hwndCtrl, bEnable);

Đối số hwndCtrl là chỉ danh của thành phần điều khiển muốn vô hiệu hóa, thành phần thứ hai là bEnable mang hai giá trị TRUE hay FALSE, nếu thành phần này mang giá trị FALSE thì điều khiển này được vô hiệu hóa, còn ngược lại nếu thành phần này mang giá trị TRUE thì điều khiển đó có hiệu hóa trở lại.

2. Hộp thoại không trạng thái

Trong phần trên đã thảo luận loại hộp thoại, thứ nhất đó là hộp thoại trạng thái, và bây giờ tiếp tục thảo luận đến loại hộp thoại thứ hai, hộp thoại không trạng thái (modeless). Để hiểu rõ cách sử dụng cũng như những thao tác trên hộp thoại không trạng thái, chúng ta thứ tự tìm hiểu qua các mục sau.

a. Sự khác nhau giữa hộp thoại trạng thái và hộp thoại không trạng thái

Hộp thoại không trạng thái khác với hộp thoại trạng thái ở chỗ. Sau khi hiển thị hộp thoại không trạng thái chúng ta có thể chuyển thao tác đến các cửa sổ khác mà không cần đóng hộp thoại dạng này lại. Điều này thuận tiện đối với người dùng khi người dùng muốn trực quan các sổ thao tác cùng một lúc. Ví dụ như ở trình soạn thảo Studio Deverloper bạn có thể thao tác qua lại giữa hai hộp thoại, đó là hộp thoại bạn cần thiết kế và một hộp thoại chứa các loại điều khiển mà bạn dùng để thiết kế. Với cách làm này giúp người dùng trực quan hơn so với cách chỉ cho phép người dùng chỉ thao tác trên một cửa sổ.

Sử dụng hàm **DialogBox** để gọi hộp thoại trạng thái và chỉ nhận được kết quả trả về khi hộp thoại này bị đóng cùng với hàm **DialogBox** kết thúc. Giá trị trả về của hàm này do đối số thứ hai của hàm kết thúc hộp thoại (**EndDialog**) quy định. Còn đối với hộp thoại không trạng thái thì được tạo ra bằng hàm.

```
hDlgModeless = CreateDialog(hInstance, szTemplate,
                            hwndParent, DialogProc);
```

Nhưng hàm này trả quyền điều khiển về cho nơi gọi ngay lập tức và giá trị trả về là handle của của hộp thoại hiện hành. Vì có thể có nhiều cửa sổ thao tác cùng một lúc nên bạn lưc handle này để dễ dàng truy cập khi bạn cần.

Phải đặt chế độ **WS_VISIBLE** cho hộp thoại không trạng thái, bằng cách chọn mục More Styles trong cửa sổ Properties của hộp thoại. Nếu như không bật chế độ **VISIBLE** lên thì chương trình phải có câu lệnh **ShowWindow** sau lời gọi hàm **CreateDialog** khi muốn hiển thị hộp thoại dạng này lên màn hình.

```
hDlgModeless = CreateDialog(hInstance, szTemplate,
                            hwndParent, DialogProc);
ShowWindow(hDlgModeless,SW_SHOW);
```

Các thông điệp gởi đến hộp thoại dạng modal do trình quản lý Windows điều khiển cũng khác với các thông điệp gởi đến hộp thoại dạng modeless phải đi qua hằng đợi của chương trình chính. Bởi vì các thông điệp của hộp thoại dạng modeless dùng chung với các thông điệp của cửa sổ chương trình chính. Như vậy chúng ta phải lọc ra thông điệp nào là thông điệp gởi đến hộp thoại khi thao tác trên hộp thoại từ trong vòng lặp nhận thông điệp. Để làm được điều này chúng ta dùng handle của hộp thoại (lưu trong biến toàn cục) được trả về từ lời gọi hàm **CreateDialog** và chuyển hướng chúng bằng đoạn lệnh như sau.

```
while (GetMessage(&msg, NULL, 0, 0)) {
    if (hDlgModeless==0 || !IsDialogMessage (hDlgModeless, &msg) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
```

```
}
```

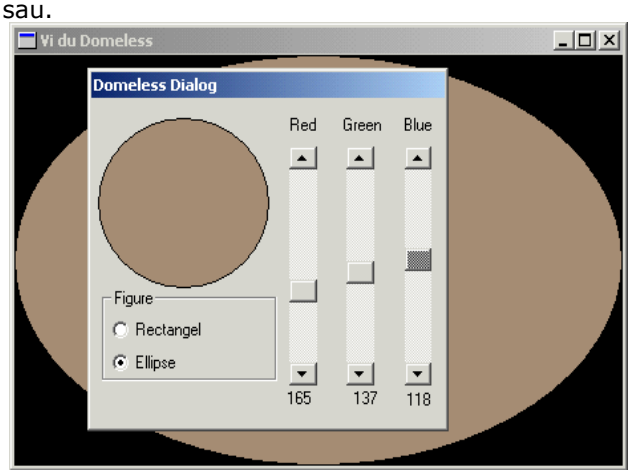
Nếu thông điệp lấy ra từ hằng đợi dành cho hộp thoại thì hàm **IsDialogMessage** kiểm tra và gởi đến các thủ tục xử lý hộp thoại. Và lúc này hàm trả về giá trị TRUE, còn ngược lại thì hàm trả về giá trị FALSE. Nếu dùng thêm chức năng phím tăng tốc thì đoạn chương trình trên được viết lại như sau.

```
while (GetMessage(&msg, NULL, 0, 0)) {
    if (hDlgModeless==0 || !IsDialogMessage(hDlgModeless, &msg) {
        if (TranslateAccelerator (hwnd, hAccel, &msg) {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }
}
```

Nên chú ý rằng biến hDlgModeless luôn mang giá trị 0 cho đến lúc có một hộp thoại được khởi tạo bằng câu lệnh **CreateDialog** thì giá trị của nó mới được thay đổi. Khi cửa sổ hộp thoại bị hủy nhớ đặt hDlgModeless về giá trị 0. Điều này giúp Windows không gởi nhầm thông điệp xử lý đến các cửa sổ khác. Để kết thúc và đóng hộp thoại dạng Modeless bạn dùng hàm **DestroyWindow** chứ không phải dùng hàm **EndDialog** như hộp thoại dạng Modal.

b. Ví dụ về hộp thoại không không trạng thái

Để minh họa cách dùng hộp thoại không trạng thái (*modeless*) ta xét ví dụ 2.3. Chương trình ví dụ 2.3 sau khi chạy có kết quả như sau.



Hình 2.6 Minh họa hộp thoại không trạng thái

Khi dùng chuột để chọn loại hình vẽ trên radio button, loại hình vẽ được chọn sẽ vẽ cùng lúc lên control tĩnh của hộp thoại và cửa sổ chính. Dùng chuột để chọn màu tô cho hình vẽ được chọn, bằng cách rê chuột lên 3 thanh cuộn **Scrollbar**.

Chương trình minh họa *(Ví dụ 2.3)*:

MODELESS.CPP *(trích dẫn)*

```
void PaintWindow (HWND hwnd, int iColor[], int iFigure) {
    HBRUSH hBrush;
    HDC hdc;
    RECT rect;
    hdc = GetDC(hwnd);
    GetClientRect (hwnd, &rect);
    hBrush = CreateSolidBrush(RGB(iColor[0], iColor[1], iColor[2]));
    hBrush = (HBRUSH) SelectObject (hdc, hBrush);
    if (iFigure == IDC_RECT)
        Rectangle (hdc, rect.left, rect.top, rect.right, rect.bottom);
    else
        Ellipse(hdc, rect.left, rect.top, rect.right, rect.bottom);
    DeleteObject (SelectObject (hdc, hBrush));
    ReleaseDC (hwnd, hdc);
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) {
```

```
    switch (message) {
        case WM_PAINT:
            PaintTheBlock(hwnd, iColor, iFigure);
            return 0;
        case WM_DESTROY:
            DeleteObject((HGDIOBJ)SetClassLong(hwnd, GCL_HBRBACKGROUND,
                        (LONG)GetStockObject(WHITE_BRUSH)));
            PostQuitMessage (0);
            return 0;
    }
    return DefWindowProc (hwnd, message, wParam, lParam);
}

void PaintTheBlock (HWND hCtrl, int iColor[], int iFigure) {
    InvalidateRect (hCtrl, NULL, TRUE);
    UpdateWindow (hCtrl);
    PaintWindow (hCtrl, iColor, iFigure);
}

BOOL CALLBACK ColorScrDlg (HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam) {
    HWND hwndParent, hCtrl;
    static HWND hCtrlBlock;
    int iCtrlID, iIndex;
    switch (message) {
        case WM_INITDIALOG:
            hCtrlBlock = GetDlgItem (hDlg, IDC_PAINT);
            for (iCtrlID = 10; iCtrlID < 13; iCtrlID++) {
                hCtrl = GetDlgItem (hDlg, iCtrlID);
                PaintTheBlock (hCtrlBlock, iColor, iFigure);
                PaintTheBlock (hwndParent, iColor, iFigure);
                SetScrollRange (hCtrl, SB_CTL, 0, 255, FALSE);
                SetScrollPos(hCtrl, SB_CTL, 0, FALSE);
            }
            return TRUE;
        case WM_COMMAND:
            switch(LOWORD(wParam)) {
                case IDC_RECT:
                case IDC_ELLIPSE:
                    iFigure = LOWORD(wParam);
                    hwndParent = GetParent (hDlg);
                    CheckRadioButton(hDlg, IDC_RECT, IDC_ELLIPSE, LOWORD (wParam));
                    PaintTheBlock(hCtrlBlock, iColor, iFigure);
                    PaintTheBlock (hwndParent, iColor, iFigure);
                    return TRUE;
            }
            break;
        case WM_VSCROLL:
            hCtrl = (HWND) lParam;
            iCtrlID = GetWindowLong (hCtrl, GWL_ID);
            iIndex = iCtrlID - 10;
            hwndParent = GetParent (hDlg);
            PaintTheBlock (hCtrlBlock, iColor, iFigure);
            PaintTheBlock (hwndParent, iColor, iFigure);
            switch (LOWORD (wParam)) {
                case SB_PAGEDOWN:
                    iColor[iIndex] += 15;
                case SB_LINEDOWN:
                    iColor[iIndex] = min (255, iColor[iIndex] + 1);
                    break;
```

```
                case SB_PAGEUP:
                    iColor[iIndex] -= 15;
                case SB_LINEUP:
                    iColor[iIndex] = max (0, iColor[iIndex] - 1);
                    break;
                case SB_TOP:
                    iColor[iIndex] = 0;
                    break;
                case SB_BOTTOM:
                    iColor[iIndex] = 255;
                    break;
                case SB_THUMBPOSITION:
                case SB_THUMBTRACK:
                    iColor[iIndex] = HIWORD (wParam);
                    break;
                default:
                    return FALSE;
            }
            SetScrollPos(hCtrl, SB_CTL, iColor[iIndex], TRUE);
            SetDlgItemInt(hDlg, iCtrlID + 3, iColor[iIndex], FALSE);
            InvalidateRect(hwndParent,NULL,TRUE);
            DeleteObject((HGDIOBJ)SetClassLong(hwndParent, GCL_HBRBACKGROUND,
                        (LONG) CreateSolidBrush(RGB(iColor[0], iColor[1],
                                                    iColor[2]))));
            return TRUE;
        case WM_PAINT:
            PaintTheBlock(hCtrlBlock, iColor, iFigure);
            break;
    }
    return FALSE;
}
```

C. MENU

Trong giao diện ứng dụng Windows, thành phần quan trọng thường không thể thiếu là **menu** của chương trình. Menu xuất hiện ngay dưới thanh tiêu đề của chương trình ứng dụng. Ngoài ra trong một số ứng dụng thanh menu có thể di chuyển được.

Thật ra menu cũng khá đơn giản, vì chúng được tổ chức thành các nhóm trên thanh chính (**File**, **Edit**, **View**,…), mỗi mục liệt kê trong menu chính có thể chứa một hay nhiều mục liệt kê gọi là **menu popup** hay **dropdown**, và với mỗi mục liệt kê trong menu popup này có thể có các mục con của nó,….

Các mục liệt kê trên menu có thể dùng để kích hoạt một lệnh, hay chọn trạng thái (**check**, **uncheck**). Các mục liệt kê trên menu có 3 dạng: có hiệu lực (**enabled**), không có hiệu lực (**disabled**), và màu xám (**grayed**). Với quan điểm lập trình thì ta chỉ cần hai trạng thái là có hiệu lực và không có hiệu lực mà thôi, do đó trạng thái màu xám sẽ chỉ cho người dùng biết là trạng thái của mục liệt kê có hiệu lực hay không. Vì vậy khi viết chương trình những mục nào không có hiệu lực thì ta thiết lập trạng thái màu xám, khi đó người dùng sẽ biết rằng mục liệt kê đó không có hiệu lực.

1. Thiết lập Menu

Để tạo một menu và đưa vào chương trình bao gồm các bước sau:

*Tạo menu trong tập tin tài nguyên ***.RC**: Để tạo menu trong tập tin tài nguyên, thường có 2 cách chính là: dùng một trình soạn thảo để mở tập tin tài nguyên và soạn thảo theo cấu trúc tập tin RC cung cấp cho tài nguyên menu. Thông thường, cách này ít sử dụng, vì các môi trường phát triển C trên Windows (*Borland C for Windows, Visual C*) đều cung cấp các công cụ cho phép tạo menu một cách dễ dàng.

*Cài đặt **menu** vào cửa sổ của chương trình ứng dụng: phần này đơn giản là khi định nghĩa lớp cửa sổ ta thiết lập thuộc tính lpszMenuName của cấu trúc lớp **WNDCLASS** bằng tên menu được khai báo trong tập tin tài nguyên.

Ví dụ: `wndclass.lpszMenuName = "MENU1";`

Ngoài ra, có thể cài đặt menu vào cửa sổ bằng cách dùng lệnh:

```
hMenu = LoadMenu (hInstance, TEXT("MENU1"));
```

Lệnh này sẽ trả về một định danh của menu được nạp, khi có được định danh menu này thì khi đưa vào cửa sổ có 2 cách sau:

- Trong hàm tạo cửa sổ **CreateWindow**, tham số thứ 9 của hàm là định danh cho menu, thiết lập tham số này là định danh của **menu** vừa tạo.

```
hwnd = CreateWindow (TEXT("MyClass"), TEXT("Window Caption"),
                     WS_OVERLAPPEDWINDOW, CW_USEDEFAULT,
                     CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
                     NULL, hMenu, hInstance, NULL);
```

- Khi gọi hàm tạo cửa sổ **CreateWindow**, tham số thứ 9 được thiết lập NULL, sau đó trong chương trình dùng lệnh: **SetMenu**(hWnd, hMenu); để thiết lập menu cho cửa sổ.

Thêm các đoạn chương trình xử lý menu: Windows phát sinh thông điệp **WM_COMMAND** và gởi đến chương trình khi người dùng chọn một mục liệt kê có hiệu lực trên thanh menu. Khi đó chỉ cần xử lý thông điệp **WM_COMMAND** bằng cách kiểm tra 16 bit thấp của tham số wParam là xác định được ID của mục liệt kê nào trên menu được chọn.

2. Ví dụ minh họa Menu

*Tập tin tài nguyên chứa khai báo menu: **MENUDEMO.RC**

```
MENUDEMO MENU DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&New", IDM_FILE_NEW
        MENUITEM "&Open", IDM_FILE_OPEN
        MENUITEM "&Save", IDM_FILE_SAVE
        MENUITEM "Save &As...", IDM_FILE_SAVE_AS
        MENUITEM SEPARATOR
        MENUITEM "E&xit", IDM_APP_EXIT
    END
    POPUP "&Edit"
    BEGIN
        MENUITEM "&Undo", IDM_EDIT_UNDO
        MENUITEM SEPARATOR
        MENUITEM "C&ut", IDM_EDIT_CUT
        MENUITEM "&Copy", IDM_EDIT_COPY
        MENUITEM "&Paste", IDM_EDIT_PASTE
        MENUITEM "De&lete", IDM_EDIT_CLEAR
    END
    POPUP "&Background"
    BEGIN
        MENUITEM "&White", IDM_BKGND_WHITE, CHECKED
        MENUITEM "&Light Gray", IDM_BKGND_LTGRAY
        MENUITEM "&Gray", IDM_BKGND_GRAY
        MENUITEM "&Dark Gray", IDM_BKGND_DKGRAY
        MENUITEM "&Black", IDM_BKGND_BLACK
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&Help...", IDM_APP_HELP
        MENUITEM "&About...", IDM_APP_ABOUT
    END
END
```

*Tập tin tiêu đề chứa các định nghĩa: **MENUDEMO.H**

```
#define IDM_FILE_NEW 40001
#define IDM_FILE_OPEN 40002
#define IDM_FILE_SAVE 40003
#define IDM_FILE_SAVE_AS 40004
#define IDM_APP_EXIT 40005
#define IDM_EDIT_UNDO 40006
```

```
#define IDM_EDIT_CUT 40007
#define IDM_EDIT_COPY 40008
#define IDM_EDIT_PASTE 40009
#define IDM_EDIT_CLEAR 40010
#define IDM_BKGND_WHITE 40011
#define IDM_BKGND_LTGRAY 40012
#define IDM_BKGND_GRAY 40013
#define IDM_BKGND_DKGRAY 40014
#define IDM_BKGND_BLACK 40015
#define IDM_APP_HELP 40018
#define IDM_APP_ABOUT 40019
```

*Tập tin chứa mã nguồn: **MENUDEMO.C**

```
#include <windows.h>
#include "menudemo.h"
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
/* Khai báo tên dùng chung cho các tài nguyên trong chương trình. */
TCHAR szAppName[] = TEXT ("MenuDemo");

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine,
int iCmdShow)
{
    HWND hwnd;
    MSG msg;
    WNDCLASS wndclass;
    wndclass.style = CS_HREDRAW | CS_VREDRAW;
    wndclass.lpfnWndProc = WndProc;
    wndclass.cbClsExtra = 0;
    wndclass.cbWndExtra = 0;
    wndclass.hInstance = hInstance;
    wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
    wndclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wndclass.lpszMenuName = szAppName;
    wndclass.lpszClassName = szAppName;
    if (!RegisterClass(&wndclass)) {
        MessageBox(NULL, TEXT("This program requires Windows "), szAppName,
                   MB_ICONERROR);
        return 0;
    }
    hwnd = CreateWindow(szAppName, TEXT("Menu Demonstration"),
                        WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT,
                        CW_USEDEFAULT, CW_USEDEFAULT, NULL, NULL, hInstance, NULL);
    ShowWindow (hwnd, iCmdShow);
    UpdateWindow (hwnd);
    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }
    return msg.wParam;
}

LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) {
    /* Khao báo danh sách các màu chổi tô, các hằng này được định nghĩa trong file
WINGDI.H */
    static int idColor[5] = {WHITE_BRUSH, LTGRAY_BRUSH, GRAY_BRUSH, DKGRAY_BRUSH,
                             BLACK_BRUSH};
    static int iSelection = IDM_BKGND_WHITE;
    HMENU hMenu;
    switch (message) {
```

```
        case WM_COMMAND:
            hMenu = GetMenu (hwnd); // Lấy định danh của menu
            switch (LOWORD (wParam)) // Kiểm tra định danh mục chọn
            {
                case IDM_FILE_NEW:
                case IDM_FILE_OPEN:
                case IDM_FILE_SAVE:
                case IDM_FILE_SAVE_AS:
                    MessageBeep(0); //Phát ra tiếng kêu bíp
                    return 0;
                case IDM_APP_EXIT:
                    /*Gửi thông điệp đóng cửa sổ lại*/
                    SendMessage (hwnd, WM_CLOSE, 0, 0);
                    return 0;
                case IDM_EDIT_UNDO:
                case IDM_EDIT_CUT:
                case IDM_EDIT_COPY:
                case IDM_EDIT_PASTE:
                case IDM_EDIT_CLEAR:
                    MessageBeep (0);
                    return 0;
                case IDM_BKGND_WHITE:
                case IDM_BKGND_LTGRAY:
                case IDM_BKGND_GRAY:
                case IDM_BKGND_DKGRAY:
                case IDM_BKGND_BLACK:
                    /* Bỏ check của mục chon trước đó*/
                    CheckMenuItem(hMenu,iSelection, MF_UNCHECKED);
                    iSelection = LOWORD (wParam); /* Lấy ID mục mới */
                    /* Check mục chọn mới*/
                    CheckMenuItem (hMenu, iSelection, MF_CHECKED);
                    /* Thiết lập màu tương ứng với mục chọn mới */
                    SetClassLong(hwnd, GCL_HBRBACKGROUND,
                       (LONG) GetStockObject(idColor[iSelection-IDM_BKGND_WHITE]));
                    InvalidateRect (hwnd, NULL, TRUE);
                    return 0;
                case IDM_APP_HELP:
                    MessageBox(hwnd, TEXT("Help not yet implemented!"), szAppName,
                               MB_ICONEXCLAMATION | MB_OK);
                    return 0;
                case IDM_APP_ABOUT:
                    MessageBox(hwnd, TEXT ("Menu Demonstration Program\n (c)
                               Charles Petzold, 1998"), szAppName,
                               MB_ICONINFORMATION | MB_OK);
                    return 0;
            }
            break;
        case WM_DESTROY:
            PostQuitMessage(0);
            return 0;
    }
    return DefWindowProc(hwnd, message, wParam, lParam);
}
```

CHƯƠNG I: CÁC ĐỐI TƯỢNG ĐIỀU KHIỂN

A. MỞ ĐẦU

Các đối tượng điều khiển (*control*) là các thành phần tương tác trực quan, thể hiện rõ cơ chế giao tiếp đồ họa giữa ứng dụng và người dùng. Nhờ các đối tượng này, các chương trình ứng dụng trong Windows trở nên thân thiện và dễ dùng. Ví thế, chúng là các thành phần cơ bản không thể thiếu trong hầu hết các ứng dụng.

Trong chương này, chúng ta sẽ tìm hiểu các tạo lập và xử lý cho các đối tượng điều khiển thông qua các lớp (*class*) sau:

- Lớp Button (nút bấm).
- Lớp Static (tĩnh).
- Lớp Edit Box (soạn thảo).
- Lớp List Box (danh sách).
- Lớp Combo Box.
- Lớp Scroll Bar (thanh cuộn).

B. GIỚI THIỆU TỔNG QUAN

Một kiểu điều khiển được xem như là một cửa sổ con. Có thể tạo nhiều cửa sổ con trong cùng một cửa sổ cha. Các cửa sổ con xác định handle cửa sổ của cha bằng cách gọi hàm:

```
hwndParent = GetParent (hwnd);
```

hwnd là handle của cửa sổ con cần lấy handle của cửa sổ cha. Và khi đã lấy được handle của cửa sổ cha, cửa sổ con có quyền gởi các thông điệp đến cửa sổ cha thông qua hàm.

```
SendMessage(hwndParent, message, wParam, lParam);
```

message là thông điệp cần gởi đến thủ tục xử lý của cửa sổ cha. `wParam` là chỉ danh `ID` của cửa sổ con, còn lParam ghi lại trạng thái của cửa sổ con.

Vậy chúng ta có thể tạo một thành phần điều khiển dạng cửa sổ con hay còn gọi là "**child window control**". Cửa sổ con có nhiệm vụ xử lý các thông điệp như bàn phím, thông điệp chuột và thông báo cho cửa sổ cha khi trạng thái của cửa sổ con thay đổi. Như vậy cửa sổ con trở thành công cụ giao tiếp (cho phép nhập và xuất) giữa người dùng với chương trình.

Tuy chúng ta có thể tạo ra một cửa sổ con cho chính mình, nhưng chúng ta nên tận dụng các lớp cửa sổ con đã được Windows định nghĩa sẵn hay còn gọi là những kiểu điều khiển chuẩn. Những kiểu điều khiển chuẩn này thường là các nút bấm (**button**), hộp kiểm tra (**check box**), hộp soạn thảo (**edit box**), hộp danh sách (**list box**), **combo box**, các thanh cuộn và chuỗi chữ. Ví dụ muốn tạo ra một nút bấm ở trên màn hình chỉ cần gọi hàm **CreateWindow**, mà chẳng cần phải quan tâm đến cách vẽ, cách nhận chuột hay là chớp khi bị kích hoạt. Tất cả điều này đều do Windows xử lý. Điều quan trọng làphải chặn thông điệp **WM_COMMAND** của các điều khiển để xử lý thông điệp này theo những mục đích khác nhau.

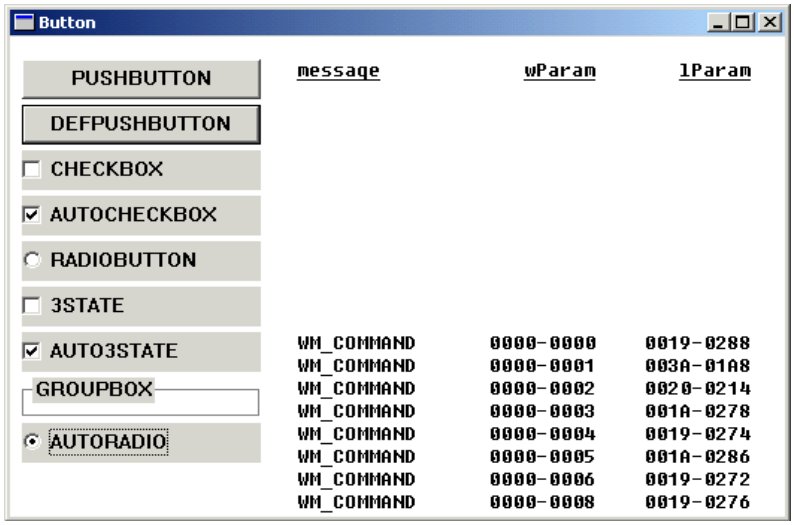
Các kiểu điều khiển con thường được dùng trong hộp thoại. Như đã minh họa trong chương 2, ở đó các điều khiển nhận hộp thoại làm cửa sổ cha. Tuy nhiên, cũng có thể tạo các kiểu điều khiển con trực tiếp trên vùng cửa sổ chính, bằng cách gọi hàm **CreateWindow** và điều chỉnh vị trí cùng với kích thước của nó cho thích hợp bằng hàm **MoveWindow**. Thủ tục xử lý thông điệp của cửa sổ cha gửi các thông điệp đến các khiểu điều khiển con, và ngược lại các child window control gởi các thông điệp để yêu cầu cửa sổ cha xử lý các thông điệp đó.

Để tạo một cửa sổ ứng dụng bình thường. Đầu tiên phải đăng ký lớp cửa sổ bằng hàm **RegisterClass**. Tiếp theo là khởi tạo lớp đã đăng ký thông qua hàm **CreateWindow**. Còn trường hợp muốn tạo một lớp đã được định nghĩa sẵn thì không cần đăng ký cho lớp cửa sổ con muốn tạo.

Sử dụng các kiểu điều khiển trực tiếp trên cửa sổ chính đòi hỏi các tác vụ cấp thấp hơn so với dùng các kiểu điều khiển trên hộp thoại. Và các kiểu điều khiển tạo ra trên cửa sổ chính không có hỗ trợ các tiện ích. Ví dụ như chúng ta không thể sử dụng phím bấm **tab** để chuyển focus giữa các kiểu điều khiển với nhau.

C. LỚP BUTTON

Để tìm hiểu các kiểu điều khiển, xem xét ví dụ 3.1 sau. Trong ví dụ này đã tạo ra 9 cửa sổ con chuẩn trên một cửa sổ cha như hình 3.1.



Hình 3.1 Minh họa các lớp Button

Nhấp chuột vào các nút, lúc đó các nút sẽ gởi thông điệp **WM_COMMAND** đến thủ tục xử lý thông điệp **WndProc** của cửa sổ cha. Thủ tục **WndProc** xử lý và in ra màn hình các thông số lParam và wParam của thông điệp gởi tới này.Trong đó lParam là handle của cửa sổ con gởi thông điệp đến cửa sổ cha. wParam có hai phần **LOWORD** và **HIWORD**, **LOWORD** cho biết ID của cửa sổ con, **HIWORD** là mã thông báo. Mã thông báo nút bấm là một trong những giá trị sau.

Định danh mã thông báo Button	Giá trị
BN_CLICKED	0
BN_PAINT	1
BN_HILETE hay BN_PUSHED	2
BN_UNHILITE hay BN_UNPHUSHED	3
BN_DISABLE	4
BN_DOUBLECLICKED hay BN_DBCLICK	5
BN_SETFOCUS	6
BN_KILLFOCUS	7

Bảng 3.1 Định danh mã thông báo Button

Không bao giờ thấy được các giá trị của nút bấm, chỉ biết rằng giá trị từ 1 đến 4 dành cho kiểu button **BS_USERBUTTON**, giá trị 5 dành cho kiểu **BS_RADIOBUTTON**, **BS_AUTORADIOBUTTON**, **BS_OWNEDRAW**, hay các nút bấm khác nếu nút bấm đó bao gồm kiểu **BS_NOTYFY**. Giá trị 6,7 dành cho các kiểu nút bấm bao gồm cả cờ **NOTIFY**. Sau đây là chương trình chính.

CONTROL1.CPP *(trích dẫn)*

```
struct {
    int iStyle;
    TCHAR *szText;
}
button[ ] = {BS_PUSHBUTTON, TEXT ("PUSHBUTTON"),
             BS_DEFPUSHBUTTON, TEXT ("DEFPUSHBUTTON"),
             BS_CHECKBOX, TEXT ("CHECKBOX"),
             BS_AUTOCHECKBOX, TEXT ("AUTOCHECKBOX"),
             BS_RADIOBUTTON, TEXT ("RADIOBUTTON"),
             BS_3STATE, TEXT ("3STATE"),
             BS_AUTO3STATE, TEXT ("AUTO3STATE"),
             BS_GROUPBOX, TEXT ("GROUPBOX"),
             BS_AUTORADIOBUTTON, TEXT ("AUTORADIO")
            };

#define NUM (sizeof(button) / sizeof(button[0]))

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
```

```
    static HWND hwndButton[NUM];
    static RECT rect;
    static TCHAR szTop[] = TEXT("message wParam lParam"),
    szUnd[] = TEXT("_______ ______ ______"),
    szFormat[] = TEXT("%-16s%04X-%04X %04X-%04X"),
    szBuffer[50];
    static int cxChar, cyChar;
    HDC hdc;
    PAINTSTRUCT ps;
    int i;
    switch (message) {
        case WM_CREATE:
            cxChar = LOWORD(GetDialogBaseUnits());
            cyChar = HIWORD(GetDialogBaseUnits());
            for (i = 0; i < NUM; i++)
                hwndButton[i] = CreateWindow(TEXT("button"), button[i].szText,
                    WS_CHILD | WS_VISIBLE | button[i].iStyle, cxChar,
                    cyChar * (1 + 2 * i), 20 * cxChar, 7 * cyChar / 4,
                    hwnd, (HMENU)i, ((LPCREATESTRUCT)lParam)->hInstance, NULL);
            return 0;
        case WM_SIZE:
            rect.left = 24 * cxChar;
            rect.top = 2 * cyChar;
            rect.right = LOWORD(lParam);
            rect.bottom = HIWORD(lParam);
            return 0;
        case WM_PAINT:
            InvalidateRect (hwnd, &rect, TRUE);
            hdc = BeginPaint (hwnd, &ps);
            SelectObject(hdc,GetStockObject(SYSTEM_FIXED_FONT));
            SetBkMode (hdc, TRANSPARENT);
            TextOut (hdc, 24 * cxChar, cyChar, szTop, lstrlen (szTop));
            TextOut (hdc, 24 * cxChar, cyChar, szUnd, lstrlen (szUnd));
            EndPaint (hwnd, &ps);
            return 0;
        case WM_DRAWITEM:
        case WM_COMMAND:
            ScrollWindow (hwnd, 0, -cyChar, &rect, &rect);
            hdc = GetDC (hwnd);
            SelectObject(hdc, GetStockObject(SYSTEM_FIXED_FONT));
            TextOut (hdc, 24*cxChar, cyChar*(rect.bottom/cyChar-1), szBuffer,
                wsprintf (szBuffer, szFormat,
                message==WM_DRAWITEM ? TEXT ("WM_DRAWITEM"): TEXT ("WM_COMMAND"),
                HIWORD (wParam), LOWORD (wParam),
                HIWORD (lParam), LOWORD (lParam)));
            ReleaseDC (hwnd, hdc);
            ValidateRect (hwnd, &rect);
            break;
        case WM_DESTROY:
            PostQuitMessage(0);
            return 0;
    }
    return DefWindowProc (hwnd, message, wParam, lParam);
}
```

Để tạo ra một child window control bạn dùng cấu trúc **CreateWindow** với các thông số như sau.

- Tên lớp: TEXT ("button")
- Text cửa sổ: Button[i].szText
- Kiểu cửa sổ: WS_CHILD | WS_VISIBLE | button[i].iStyle
- Vị trí x: cxChar

- Vị trí y: cyChar*(1+2*i)
- Chiều rộng: 20*xChar
- Chiều cao: 7*yChar*4
- Handle cửa sổ cha: hwnd
- Chỉ danh của cửa sổ con: (HMENU) i
- Thẻ quản Handle: ((LPCREATESTRUCT) lParam-> hInstance, NULL);
- Các thông số thêm: NULL

Trong đó tên lớp là cố định. Tên cửa sổ do chúng ta đặt. Kiểu cửa sổ sử dụng là **WS_CHILD**, **WS_VISIBLE** và một trong 9 kiểu button (**BS_PUSHBUTTON**, **BS_CHECKBOX**, …).

Tiếp theo là 4 thông số xác định ví trí **x**, ví trí **y**, kích thước theo chiều rộng, kích thước chiều cao của cửa sổ con trên vùng client của cửa sổ cha. hwnd là handle của cửa sổ cha. ID là chỉ danh của mỗi cửa sổ con (mỗi cửa sổ con có duy nhất mỗi số ID). ID này phải ép kiểu **HMENU** để chỉ định trình đơn. lParam thực chất là một con trỏ đến cấu trúc **LPCREATESTRUCT** có thành phần hInstance. Dó đó muốn lấy thẻ quản hInstance thì phải ép kiểu lParam.

1. Lớp Push Button

Trong ví dụ 3.1 có hai **Push Button** được tạo ra bằng hàm **CreateWindow** với kích thước và ví trí được xác định bởi người lập trình. Các Push Button được sử dụng để bật tắt một hành động tức thời chứ không giữ được trạng thái bật hay tắt lâu dài như checkbox được. Trên đây là hai kiểu cửa sổ **BS_PUSHBUTTON** và **BS_DEFBUTTON** (kiểu nút bấm mặc định). Hai kiểu này khi thiết kế thì khác nhau nhưng khi sử dụng thì nó có chức năng hoàn toàn giống nhau. Khi nhấn chuột vào nút này thì nút này gởi thông điệp **WM_COMMAND** đến cửa sổ cha với mã thông báo **BN_CLICK**. Có thể tác động đến nút bấm này bằng cách gọi hàm.

```
SendMessage(hwndButton, BM_SETSTASE, 1, 0);
```

Nếu muốn nút nhấn này trở lại trạng thái bình thường thì gọi hàm:

```
SendMessage(hwndButton, BM_SETSTASE, 0, 0);
```

`hwndButton` là định danh của cửa sổ con được trả về bởi hàm **CreateWindow**.

2. Lớp Check Box

Một **check box** là một hộp vuông kèm theo chữ. Thông thường chữ nằm ở bên trái của hộp. Tuy nhiên, cũng có thể đặt chữ nằm ở bên phải bằng cách thêm vào kiểu **BS_LEFTTEXT** khi tạo một **button**. Các check box cho phép người dùng chọn các tùy chọn, nó hoạt động như một công tắc. Có hai loại check box thông dụng đó là **BS_CHECKBOX** và **BS_AUTOCHECKBOX**. Khi sử dụng loại **BS_CHECKBOX**, chúng ta tự đặt dấu check box bằng cách gởi đến kiểu điều khiển này thông điệp **BS_SETCHECK**. Thông số wParam trong hàm **SendMessage** được đặt giá trị 1 để tạo đánh dấu, và bằng 0 khi muốn hủy đánh dấu. Lấy trạng thái của một check box bằng cách gởi đến kiểu điều khiển này thông điệp **BM_GETCHECK**. Dùng đoạn chương trình sau để bật tắt dấu check khi xử lý thông điệp **WM_COMMAND** được gởi đến từ các kiểu điều khiển.

```
SendMessage((HWND)lParam, BM_SETCHECK,
            (WPARAM)!SendMessage((HWND)lParam, BM_GETCHECK, 0, 0), 0);
```

Chú ý toán tử **!** (NOT) đứng trước hàm **SendMessage**. Giá trị lParam là handle của cửa sổ con gởi đến cửa sổ cha trong thông điệp **WM_COMMAND**. Muốn biết trạng thái của check box nào đó thì gởi tới nó thông điệp **BM_GETCHECK**. Để khởi động một check box loại **BS_CHECKBOX** với trạng thái được đánh dấu, bằng cách gởi đến nó một thông điệp **BM_SETCHECK** theo cấu trúc.

```
SendMessage (hwndButton,BM_SETCHECK, 1, 0);
```

Còn check box **BS_AUTOCHECK** là loại nút bấm mà tự nó đánh dấu bật hay tắt cho chính nó. Muốn lấy trạng thái của check box hiện hành, chỉ cần gởi thông điệp **BM_GETCHECK** đến kiểu điều khiển này theo cấu trúc.

```
iCheck = SendMessage (hwndButton, BM_SETCHECK, 1, 0);
```

`iCheck` mang giá trị `TRUE` nếu check box ở trạng thái chọn, còn ngược lại `iCheck` mang giá trị `FALSE`.

Ngoài ra còn có hai loại check box khác là **BS_3STATE** và **BS_AUTO3STATE**. Hai loại này còn có thêm trạng thái thứ 3, đó là trạng thái nút check box có màu xám xuất hiện khi bạn gởi thông điệp **WM_SETCHECK** với tham số wParam bằng 2 đến check box này. Màu xám cho biết người dùng chọn lựa không thích hợp hay không xác định.

3. Lớp Radio Button

Một **radio button** là một vòng tròn có kèm theo chữ. Tại một thời điểm chỉ có một radio button được nhấn. Các radio thường được nhóm lại để sử dụng cho việc lựa chọn duy nhất trong nhóm. Trạng thái các radio button không bật tắt như check box. Có nghĩa, khi nhấn chuột vào radio button thì button này được đánh dấu, và khi ta nhấn chuột vào một lần nữa thì radio đó cũng vẫn ở trạng thái đánh dấu. Có hai kiểu radio button là **BS_RADIOBUTTON** và **BS_AUTORADIOBUTTON**, nhưng kiểu thứ hai chỉ sử dụng trong hộp thoại.

Khi nhận thông điệp **WM_COMMAND** từ radio button, thì chúng ta phải đánh dấu radio đó bằng cách gởi thông điệp **BM_SETCHECK** với thông số `wParam` bằng `1` như sau.

```
SendMessage(hwndButton, BM_SETCHECK, 1, 0);
```

Tất cả các radio button trong cùng một nhóm, nếu bạn muốn tắt dấu check thì bạn gởi đến chúng thông điệp **BM_SETCHECK** với thông số `wParam` bằng `0` như sau.

```
SendMessage(hwndButton, BM_SETCHECK, 0, 0);
```

4. Lớp Group Box

Group box có kiểu **BS_GROUPBOX**, đây là loại button đặc biệt. Một group box chỉ đơn giản là một đường viền có dòng tiêu đề ở trên đỉnh. Group box không xử lý các thông điệp bàn phím, không xử lý các thông điệp chuột và cũng không gởi thông điệp **WM_COMMAND** đến cửa sổ cha của nó. Các group box thường được sử dụng bao quanh các kiểu điều khiển khác.

D. LỚP STATIC

Tạo ra một lớp tĩnh bằng cách sử dụng "static" khi tạo lớp cửa sổ trong hàm **CreateWindow**. Lớp tĩnh không nhận nhập dữ liệu từ bàn phím cũng như từ chuột, và không gởi thông điệp **WM_COMMAND** đến cửa sổ cha.

Khi di chuyển hay nhấn chuột vào các cửa sổ con tĩnh, cửa sổ con này bẫy thông điệp **WM_NCHITTEST** và trả về giá trị **HTTRANSPARENT** đến Windows. Điều này làm cho Windows gởi cùng thông điệp **WM_NCHITTEST** cho cửa sổ cha. Cửa sổ cha thường gởi thông điệp này đến thủ tục **DefWindowProc**. Các kiểu cửa sổ tĩnh sau đây dùng để vẽ một hình chữ nhật hay một khung lên vùng client của cửa sổ con. Các kiểu **FRAME** là những đường bao hình chữ nhật, các kiểu **RECT** là những hình chữ nhật:

```
SS_BLACKRECT, SS_GRAYRECT, SS_ WHITERECT.
SS_BLACKFRAME, SS_GRAYFAME, SS_WHITEFRAME.
```

E. LỚP EDIT TEXT

Trong một phương diện nào đó thì lớp soạn thảo (**edit text**) được xem là một cửa sổ được định nghĩa sẵn đơn giản nhất. Nhưng xét một khía cạnh khác thì nó lại phức tạp nhất. Dùng tên lớp "edit" cùng với các thông số ví trí **x**, vị trí **y**, chiều rộng, chiều cao trong hàm **CreateWindow** để tạo ra cửa sổ soạn thảo. Khi cửa sổ soạn thảo nhận focus thì chúng ta có thể gõ chữ vào, xoá các chữ, đánh dấu các chữ, … Các thao tác trên được Windows hỗ trợ hoàn toàn.

Một trong những ứng dụng thường xuyên nhất, và đơn giản nhất của lớp soạn thảo là tạo ra một cửa sổ cho phép người dùng nhập các chữ vào. Để minh họa cho cửa sổ nhập ta xét ví dụ 3.2 sau.

EDITTEXT.CPP

```
#include <windows.h>
#define ID_EDIT 1
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
TCHAR szAppName[] = TEXT ("PopPad1");
LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    static HWND hwndEdit;
    switch (message) {
        case WM_CREATE:
            hwndEdit = CreateWindow (TEXT("edit"), NULL, WS_CHILD | WS_VISIBLE |
WS_HSCROLL | WS_VSCROLL | WS_BORDER | ES_LEFT | ES_MULTILINE | ES_AUTOHSCROLL |
ES_AUTOVSCROLL, 0, 0, 0, 0, hwnd, (HMENU)ID_EDIT, ((LPCREATESTRUCT) lParam) ->
hInstance, NULL);
            return 0;
        case WM_SETFOCUS:
            SetFocus (hwndEdit);
```

```
            return 0;
        case WM_SIZE:
            MoveWindow (hwndEdit, 0, 0, LOWORD(lParam), HIWORD(lParam), TRUE);
            return 0;
        case WM_COMMAND:
            if (LOWORD (wParam) == ID_EDIT)
                if (HIWORD(wParam)==EN_ERRSPACE || HIWORD(wParam)==EN_MAXTEXT)
                    MessageBox(hwnd, TEXT("Edit control out of space."), szAppName,
MB_OK | MB_ICONSTOP);
            return 0;
        case WM_DESTROY:
            PostQuitMessage(0);
            return 0;
    }
    return DefWindowProc (hwnd, message, wParam, lParam);
}
```

Hạn chế của edit box được định nghĩa sẵn là số ký tự người dùng nhập vào phải không quá 30.000 ký tự chữ.

1. Các kiểu lớp Edit Text

Trong ví dụ trên đã tạo một edit box bằng cách gọi hàm **CreateWindow**. Có kiểu cửa sổ là **WS_CHILD**, cùng một số tùy chọn. Chúng ta có thể canh trái, phải, giữa các chữ trong vùng cửa sổ edit box bằng cách thay đổi thông số kiểu cửa sổ **ES_LEFT**, **ES_RIGHT**, **ES_CENTER** trong hàm **CreateWindow**.

Có thể tạo một edit control cho phép hiển thị nhiều hàng bằng cách chọn kiểu cửa sổ **ES_MULTILINE**. Một edit control mặc định chỉ cho phép nhập một hàng ký tự cho đến cuối edit box. Sử dụng **ES_AUTOHSCROLL**, **ES_AUTOVSCROLL** để tạo một edit control có thanh cuộn ngang, và cuộn đứng tự động. Có thể thêm thanh cuộn ngang và đứng vào edit control bằng cách sử dụng kiểu cửa sổ **WS_HSCROLL**, **WS_VSCROLL**. Dùng kiểu cửa sổ **WS_BORDER** để tạo đường viền cho edit control.

Kích thước của edit control được xác định bằng cách gọi hàm **MoveWindow** khi hàm **WndProc** xử lý thông điệp **WM_SIZE**. Trong ví dụ trên thì kích thước của edit control được đặt bằng kích thước của cửa sổ chính.

```
MoveWindow(hwndEdit, 0, 0, LOWORD(lParam), HIWORD(lParam), TRUE);
```

Các edit control gởi thông điệp **WM_COMMAND** cùng các thông số `wParam`, `lParam` đến window cửa sổ cha. Với ý nghĩa: **LOWORD**`(wParam)` là ID cửa sổ con, **HIWORD**`(wParam)` là mã thông báo. lParam là handle của edit control.

Mã thông báo	Ý nghĩa
EN_SETFORCUS	Edit control nhận focus nhập.
EN_KILLFORCUS	Edit control mất focus nhập.
EN_CHANGE	Nội dung của edit control sẽ thay đổi.
EN_UPDATE	Nội dung của edit control thay đổi.
EN_ERRSPACE	Edit control chạy quá thời gian.
EN_MAXTEXT	Edit control chạy quá không gian khi chèn.
EN_HSCROLL	Thanh cuộn đứng của edit control bị tác động.
EN_VSCROLL	Thanh cuộn nằm của edit control bị tác động.

Bảng 3.2 Danh sách mã thông báo của edit Control.

2. Các thông điệp đến một lớp Edit Text

Các thứ tự thông điệp sau cho phép cắt, sao chép, xoá các phần chữ được chọn (*selected*).

```
SendMessage (hwndEdit, WM_CUT, 0, 0);
SendMessage (hwndEdit, WM_COPY, 0, 0);
SendMessage (hwndEdit, WM_CLEAR, 0, 0);
```

Với `WM_CUT` cắt phần chữ đã được đánh dấu đưa vào vùng Clipboard. `WM_COPY` sao chép phần chữ đã được đánh dấu đưa vào Clipboard nhưng phần đánh dấu vẫn còn trên edit control. `WM_CLEAR` xóa phần chữ đã được đánh dấu mà không đưa vào clipboard.

Chèn phần chữ nằm trong clipboard vào vùng soạn thảo edit control bằng cách gọi hàm.

```
SendMessage (hwndEdit, WM_PASTE, 0, 0);
```

Nhận bắt đầu và kết thúc của phần chữ đã chọn bằng cách gọi hàm:

SendMessage (hwndEdit, EM_GETSEL, (**WPARAM**)&iStart, (**LPARAM**)&iEnd);

iStart lưu vị trí bắt đầu và iEnd lưu ví trí kết thúc.

Để thay thế phần chữ đã chọn bằng chữ khác, ta dùng hàm;

SendMessage(hwndEdit,EM_REPLACESEL,0,(**LPARAM**)szString);

Trong đó szString là chuỗi muốn thay thế.

Đối với edit control nhiều dòng, ta đếm số dòng chữ bằng hàm.

iCount = **SendMessage** (hwndEdit, EM_GETLINECOUNT, 0, 0);

Các dòng trong edit control được đánh số bắt thứ tự từ 0. Lấy chiều dài của một dòng bằng lệnh.

iLength = **SendMessage** (hwndEdit, EM_LINELENGTH, iLine, 0).

Chép hàng này vào bộ đệm bằng cách gọi hàm.

iLength = **SendMessage** (hwdEdit, EM_GETLINE, iLine, (**LPARAM**)Buffer).

F. LỚP LIST BOX

List box là tập hợp các chuỗi kí tự được gói gọn trong một hình chữ nhật. Một chương trình có thể thêm hoặc xóa các chuỗi trong list box bằng cách gởi các thông điệp đến thủ tục window của list box. List box control gởi thông điệp **WM_COMMAND** đến cửa sổ cha khi có một mục trong list box bị đánh dấu. Cửa sổ cha xác nhận các mục trong list box đã bị đánh dấu.

Một list box có thể chọn được một mục hay nhiều mục cùng một lúc (tùy theo loại list box đơn hay kép).

1. Các kiểu List Box

Chúng ta tạo một cửa sổ con list box bằng hàm **CreateWindow** với lớp cửa sổ là "listbox" cùng với loại cửa sổ **WS_CHILD**. Tuy nhiên kiểu cửa sổ con mặt định này không gởi thông điệp **WM_COMMAND** đến cửa sổ cha, có nghĩa chương trình tự kiểm tra việc đánh dấu các danh mục trong list box. Vì thế, các kiểu điều khiển list box thường định nghĩa kiểu list box **LBS_NOTYFY**, điều này cho phép cửa sổ cha nhận thông điệp **WM_COMMAND** từ list box. Nếu muốn sắp xếp các mục trong list box thì sử dụng kiểu **LBS_SORT**.

Theo mặc định, những list box tạo ra là những list box đơn. Vì thế, nếu muốn tạo ra một list box kép (tức list box cho phép người dùng chọn nhiều dòng cùng lúc) thì phải sử dụng loại list box **LBS_MULTIPLESEL**. Thông thường, List box sẽ tự cập nhật khi một mục được thêm vào. Tuy nhiên có thể ngăn cản việc cập nhật này bằng kiểu **LBS_NOREDRAW**. Việc làm này đôi khi không thích lắm, thay vào đó chúng ta có thể sử dụng thông điệp **WM_SETREDRAW** để ngăn chặn tạm thời việc vẽ lại của list box.

Theo mặc định, các mục trong list box không có đường viền bao quanh khi hiển thị trên màn hình. Tuy nhiên, có thể thêm đường viền cho các mục bằng định danh cửa sổ **WS_BORDER**. Thêm thanh cuộn đứng vào list box bằng cách thêm định danh cửa sổ **WS_VSCROLL**.

Thông thường windows định nghĩa một list box gồm các thông số sau: LBS_NOTIFY | LBS_SORT | WS_VSCROLL | WS_BORDER.

Ngoài ra còn có các thông số **WS_SIZEBOX** và **WS_CAPTION** dùng để thay đổi kích thước và thêm tiêu đề cho list box. Nên tạo ra một list box có chiều rộng bằng chiều dài của chuỗi dài nhất trong list box cộng với chiều dài của thanh cuộn dọc. Chiều dài của thanh cuộn dọc được xác định bằng hàm.

GetSystemMetrics(SM_CXVCROLL);

2. Đặt các chuỗi vào List Box

Để thêm một chuỗi vào list box ta gởi các thông điệp đến thủ tục Windows list box bằng hàm **SendMessage**. Khi truyền chuỗi cho hàm **SendMessage** thì thông số wParam là một con trỏ, trỏ đến chuỗi được kết thúc bởi ký tự NULL. Hàm **SendMessage** trả về mã **LB_ERRSPACE** (giá trị -2) khi Windows chạy quá không gian bộ nhớ dành cho list box. Hàm **SendMessage** trả về mã **LB_ERR** (-1) khi xảy ra lỗi khác và trả về mã **LB_OKAY** (0) nếu thao tác thêm thành công.

Nếu sử dụng kiểu **LBS_SORT** thì khi thêm một chuỗi vào list box chỉ cần dùng chỉ thị **LB_ADDSTRING** theo cấu trúc.

SendMessage(hwndList, LB_ADDSTRING, 0, (**LPARAM**) szString);

Với szString là chuỗi cần thêm vào list box.

Nếu trong list box không dùng kiểu **LBS_SORT** thì có thể chèn một chuỗi với chỉ thị **LB_INSERTSTRING** cùng vị trí muốn chèn bằng hàm.

SendMessage(hwndList, LB_INSERTSTRING, iIndex, (**LPARAM**) szString);

iIndex là vị trí muốn chèn chuỗi vào. Nếu giá trị này bằng -1 thì chuỗi được chèn vào đáy của list box.

Xóa một chuỗi trong list box bằng chỉ thị **LB_DELETESTRING**.

```
SendMessage(hwndList, LB_DELETESTRING, iIndex, 0);
```

Xóa hết các phần tử nằm trong list box thì dùng chỉ thị **LB_RESETCONTENT** với cấu trúc.

```
SendMessage(hwndList, LB_LB_RESETCONTENT, 0, 0);
```

Khi thêm vào hay xóa thì Windows tự cập nhật lại list box. Tuy nhiên ta cũng có thể tạm thời cản sự cập nhật này bằng cách tắt cờ vẽ lại list box.

```
SendMessage(hwndList, WMSETREDRAW, FALSE, 0);
```

Sau khi thực hiện xong ta bật cờ vẽ lại list box bằng hàm.

```
SendMessage(hwndList, WMSETREDRAW, TRUE, 0);
```

3. Chọn và lấy các mục trên List Box

Tương tự như đặt chuỗi vào List box, chọn và lấy mục trong List box cũng phải gởi các thông điệp đến thủ tục Window List box bằng hàm **SendMessage**.

Dùng chỉ thị **LB_GETCOUNT** để đếm số mục trong List box.

```
iCount = SendMessage (hwndList, LB_GETCOUNT, 0, 0);
```

Làm sáng mục chọn mặc định thì dùng **LB_SETCURSEL**.

```
SendMessage (hwndList, LB_SETCURSEL, iIndex, 0);
```

Nếu đặt giá trị `iIndex` bằng `-1` thì window sẽ bỏ tất cả các mục chọn. Để chọn các mục dựa trên chữ bắt đầu của mục, ta dùng hàm.

```
iIndex = SendMessage (hwndlist, LB_SELECTSTRING, iIndex,
                      (LPARAM) szSearchString);
```

`iIndex` là vị trí bắt đầu của việc tìm với kí tự đầu giống `szSearchString`. Nếu giá trị `iIndex` bằng -1 thì việc tìm bắt đầu từ vị trí đầu tiên. Hàm sẽ trả về giá trị tìm đượci. Nếu chuỗi không tồn tại thì hàm trả về mã lỗi **LB_ERR**.

Xác định mục đã chọn khi nhận được thông điệp **WM_COMMAND** từ list box bằng hàm.

```
iIndex = SendMessage(hwndList, LB_GETCURSEL, 0, 0);
```

Hàm trả về vị trí của mục được chọn, còn ngược lại nếu không có mục nào được chọn thì hàm trả về mã lỗi **LB_ERR**.

Muốn xác định chiều dài của một chuỗi bất kỳ có trong list box dùng hàm.

```
iLength = SendMessage(hwndList, LB_GETTEXTLEN, iIndex, 0);
```

Với `iIndex` là ví trí của chuỗi cần xác định chiều dài. Để chép chuỗi trên vào vùng đệp Buffer, ta dùng hàm.

```
iLength = SendMessage(hwndList, LB_GETTEXT, iInde, (LPARAM) Buffer);
```

Giá trị trả về của hai hàm trên là chiều dài của chuỗi ký tự. Nên định kích thước vùng buffer sao cho đủ chứa chiều dài của chuỗi cộng thêm ký tự kết thúc chuỗi cần ghi vào.

Tuy nhiên, đối với list box chọn kép thì chúng ta không thể dùng các chỉ thị **LB_SETCURSEL**, **LB_GETCURSEL**, hoặc **LB_SELECTSTRING**. Thay vào đó phải dùng chỉ thị **LB_SETSEL** để chọn một mục mà không làm ảnh hưởng đến các mục đã chọn khác.

```
SendMessage(hwndList, LB_SETSEL, wParam, iIndex);
```

Tham số `wParam` khác 0 để chọn và làm sáng mục, bằng 0 để hủy việc chọn.

Xác định trạng thái của một mục nào đó trong list box (loại list box chọn kép) dùng hàm.

```
iSelect = SendMessage(hwndList, LB_GETSEL, iIndex, 0);
```

Hàm trả về giá trị khác 0 nếu mục có vị trí `iIndex` được chọn, và bằng 0 nếu mục ở vị trí đó không được chọn.

4. Nhận các thông điệp từ List Box

Khi dùng chuột nhấn vào list box, khi đó list box nhận focus nhập. Cửa sổ cha có thể đặt focus nhập đến list box bằng hàm.

```
SetFocus (hwndList);
```

Khi list box nhận focus nhập, chúng ta dùng con chuột, các phím chữ, phím **Spacebar** để chọn các mục trong list box. List box gởi thông điệp **WM_COMMAND** với các thông số wParam, lParam đến cửa sổ cha với ý nghĩa:

`LOWORD (wParam)` ID cửa sổ con.

`HIWORD (wParam)` Mã thông báo.

`lParam` Handle cửa sổ con.

Mã thông báo	Giá trị	Ý nghĩa
LBN_ERRSPACE	-2	Con trol list box chạy quá không gian

LBN_SELCHANGE	1	Cho biết mục chọn hiện hành đã thay đổi
LBN_DBLCLK	2	Cho biết một mục đã bị double click với chuột
LBN_SELCANCEL	3	Cho biết người dùng thay đổi mục chọn trong list box
LBN_SETFOCUS	4	Cho biết list box đang nhận được focus nhập
LBN_KILLFOCUS	5	Cho biết list box đã mất focus nhập

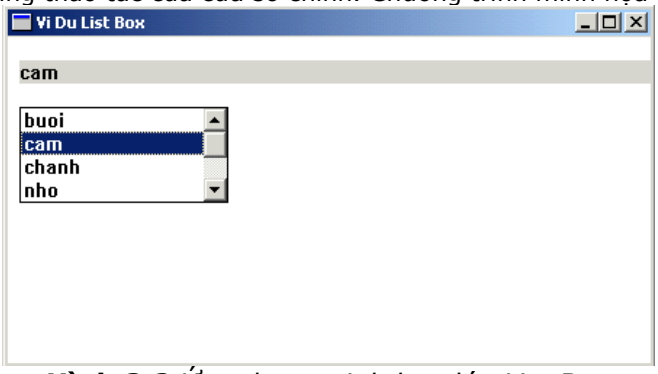
Bảng 3.3 Các giá trị của mã thông báo

Mã thông báo **LBN_ERRSPACE**, **LBN_SELCHANGE** được gởi đến cửa sổ cha khi kiểu cửa sổ list box bao gồm **LBS_NOTIFY**.

5. Một ứng dụng List Box

Sau đây là một ví dụ về list box, kết quả chương trình sau khi chạy thể hiện trong hình 3.2.

Trong ví dụ này, ta tạo ra một list box gồm các phần tử **cam**, **chanh**, **nho**…. Khi dùng chuột hay dùng phím Spacebar cộng với phím mũi tên để chọn các mục trong list box, mục được chọn sẽ có màu tô thể hiện được chọn lựa (mặc định trong Windows là màu xanh) và nội dung mục chọn này được hiển thị lên kiểu điều khiển tĩnh nằm bên trên vùng thao tác của cửa sổ chính. Chương trình minh họa như sau (ví dụ 3.2).



Hình 3.2 Ứng dụng minh họa lớp List Box

LISTBOX.CPP *(trích dẫn)*

```
#include <windows.h>
#define ID_LIST 1
#define ID_TEXT 2
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
void FillListBox (HWND hwndList) {
    int i;
    static TCHAR *tc[] = {TEXT("cam"), TEXT("quyt"), TEXT("buoi"), TEXT("tao"),
TEXT("chanh"), TEXT("xoai"), TEXT("nho")};
    for (i = 0; i < 7; i++)
        SendMessage(hwndList, LB_ADDSTRING, 0, (LPARAM)tc[i]);
}

LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    static HWND hwndList, hwndText;
    int iIndex, iLength, cxChar, cyChar;
    TCHAR *pVarName;
    switch (message) {
        case WM_CREATE:
            cxChar = LOWORD (GetDialogBaseUnits ());
            cyChar = HIWORD (GetDialogBaseUnits ());
            // Tạo một listbox
            hwndList = CreateWindow(TEXT("listbox"), NULL,
                                    WS_CHILD | WS_VISIBLE | LBS_NOTIFY | LBS_STANDARD,
                                    cxChar, cyChar * 3,
                                    cxChar * 16 + GetSystemMetrics (SM_CXVSCROLL),
                                    cyChar * 5, hwnd, (HMENU) ID_LIST,
                                    (HINSTANCE)GetWindowLong(hwnd, GWL_HINSTANCE), NULL);
            // tạo một control "static"
            hwndText = CreateWindow (TEXT("static"), NULL,
```

```
                                WS_CHILD | WS_VISIBLE | SS_LEFT, cxChar, cyChar,
                                GetSystemMetrics (SM_CXSCREEN), cyChar, hwnd,
                                (HMENU) ID_TEXT,
                                (HINSTANCE)GetWindowLong(hwnd, GWL_HINSTANCE), NULL);
            FillListBox (hwndList);
            return 0;
        case WM_SETFOCUS: //đặt focus nhập cho list box
            SetFocus (hwndList);
            return 0;
        //xử lý các thông điệp khi chọn mục trong list box
        case WM_COMMAND:
            if (LOWORD (wParam) == ID_LIST && HIWORD (wParam) == LBN_SELCHANGE) {
                // lấy vị trí của mục được chọn
                iIndex = SendMessage(hwndList,LB_GETCURSEL, 0, 0);
                iLength = SendMessage(hwndList, LB_GETTEXTLEN, iIndex, 0) + 1;
                pVarName =(char*) calloc (iLength, sizeof (TCHAR));
                SendMessage(hwndList, LB_GETTEXT, iIndex, (LPARAM)pVarName);
                SetWindowText(hwndText, pVarName);
                free(pVarName);
            }
            return 0;
        case WM_DESTROY:
            PostQuitMessage (0);
            return 0;
    }
    return DefWindowProc (hwnd, message, wParam, lParam);
}
```

Khi dùng chuột hay bàn phím chọn mục trong list box, thì list box gởi thông điệp **WM_COMMAND** cho thủ tục **WndProc** của cửa sổ cha xử lý. Khi thủ tục **WndProc** nhận được thông điệp **WM_COMMAND**, nó kiểm tra word thấp của tham số wParam có bằng định danh của list box (ở ví dụ trên là ID_LIST) không và word cao của tham số wParam(mã thông báo) có bằng **LBN_SELCHANGE** không. Nếu bằng, nó lấy chỉ số của mục được chọn thông qua hàm:

```
SendMessage (hwndList, LB_GETCURSEL, 0, 0);
```

Và tiếp tục lấy nội dung của mục được chọn thông qua hàm.

```
SendMessage (hwndList, LB_GETTEXT, iIndex, (LPARAM) pVarName);
```

Sau cùng, hiển thị mục được chọn lên control tĩnh thông qua hàm.

```
SetWindowText (hwndText, pVarName);
```

G. LỚP COMBO BOX

Cũng giống như List box, **Combo box** cũng là tập hợp các chuỗi kí tự được gói gọn trong một hình chữ nhật. Một chương trình có thể thêm hoặc xóa các chuỗi trong Combo box bằng cách gởi các thông điệp đến thủ tục window của Combo box. Combo box control gởi thông điệp **WM_COMMAND** đến cửa sổ cha khi có một mục trong Combo box được đánh dấu. Cửa sổ cha xác nhận mục được đánh dấu trong Combox box.

Trong Combox box chỉ cho phép chọn một mục trong danh sách các mục có trong Combo box.

1. Sự khác và giống nhau giữa Combo Box và List Box

Các thao tác trên list box và combo box là hoàn toàn giống nhau, chỉ khác nhau ở hình thức thể hiện. List box hiển thị các mục khi chúng được cập nhật, còn combo box chỉ hiển thị các mục khi ta nhấp chuột vào nó. Khi tạo kiểu điều khiển combo box với hàm **CreateWindow** phải sử dụng lớp cửa sổ "combobox" cùng với kiểu cửa sổ là **WS_CHILD**.

List box có thể cho phép chọn nhiều mục cùng lúc, còn combo box thì chỉ cho phép chọn một mục mà thôi.

Không giống như list box, combo box có hai phần. Phần dưới là danh sách các mục được thêm vào combo box, phần trên là một edit control dùng để hiển thị mục chọn hiện hành.

2. Thêm, xóa một chuỗi trên Combo Box

Để thêm một chuỗi vào combo box dùng hàm **SendMessage** với thông điệp **CB_ADDSTRING**:

```
SendMessage (hWnd, CB_ADDSTRING, 0, (LPARAM) szString);
```

hWnd là handle của Combo box control, szString là chuỗi cần thêm vào.

Hạn chế chiều dài của chuỗi cần thêm vào bằng thông điệp **CB_LIMITTEXT**.

SendMessage (hWnd, CB_LIMITTEXT, (**WPARAM**)length, 0).

length là chiều dài của chuỗi cần thêm vào không kể kí tự kết thúc chuỗi. Để chèn một chuỗi vào Combo box tại ví trí xác định, ta dùng thông điệp CB_INSERTSTRING.

SendMessage (hWnd, CB_INSERTSTRING, iIndex, (**LPARAM**) szString).

iIndex là vị trí cần chèn chuỗi vào Combo box. Để xóa tất cả các mục trong Combo box, ta dùng thông điệp **CB_RESETCONTENT**.

SendMessage (hWnd, CB_RESETCONTENT, 0, 0).

Tương tự, ta có thể xóa một mục trong Combo box bằng thông điệp **CB_DELETESTRING**.

SendMessage (hWnd, CB_DELETESTRING, iIndex, 0);

3. Chọn và lấy mục trên Combo box

Đếm số mục có trong Combo box bằng thông điệp **CB_GETCOUNT**.

iCount = **SendMessage** (hWnd, CB_GETCOUNT, 0, 0).

Hàm trên trả về số mục có trong Combo box. Thông thường ta chọn mục từ Combo box. Tuy nhiên, ta cũng có thể chọn mục bằng thông điệp **CB_SETCURSEL**.

SendMessage (hWnd, CB_SETCURSEL, (WPARAM) iIndex, 0).

Với iIndex là vị trí cần chọn, nếu giá trị này bằng –1 thì Windows loại bỏ việc chọn đối với tất cả các mục trong Combo box.

Khi nhận thông điệp **WM_COMMAND** từ Combo box, chúng ta có thể xác định mục được chọn có ví trí thứ mấy trong Combo box (vị trí bắt đầu là 0) bằng cách dùng thông điệp **CB_GETCURSEL**.

```
iIndex = SendMessage (hWnd, CB_GETCURSEL, 0, 0);
```

iIndex là vị trí của mục được chọn. Nếu không có mục nào được chọn thì hàm trên sẽ trả về mã lỗi **CB_ERR** (giá trị bằng –1).

Để xác định chiều dài iLength của một chuỗi có trong Combo box, ta dùng thông điệp **CB_GETLBTEXTLEN**.

```
iLength = SendMessage (hwnd, CB_GETLBTEXTLEN, iIndex, 0);
```

Hàm trả về chiều dài chuỗi với iIndex là ví trí của chuỗi cần lấy chiều dài. Nếu muốn lấy nội dung của mục nào đó, ta sử dụng thông điệp **CB_GETLBTEXT**.

```
SendMessage (hWnd, CB_GETLBTEXT, iIndex, (LPARAM)szString);
```

iIndex là vị trí chuỗi cần lấy, szString dùng để chứa chuỗi lấy được.

4. Một ứng dụng Combo box

Để minh họa cách cách sử dụng Combo box cùng với các thủ tục xử lý thông điệp trong Combo box chúng ta cùng nhau khảo sát ví dụ sau (ví dụ 3.3). Kết quả thực hiện chương trình.

Hình 3.3 Ứng dụng minh họa lớp Combo Box

COMBOBOX.CPP *(trích dẫn)*

```
void FillCombo(HWND hwndCombo) {
    int i;
```

```
    static TCHAR *tc[ ] = {TEXT("MAI XUAN HUNG"), TEXT("LE HOAN VU"),
                           TEXT("LE LU NHA"), TEXT("PHAM THANH PHONG"),
                           TEXT("LE LUC"), TEXT("NGUYEN TIEN"),
                           TEXT("DINH QUYEN")};
    for(i = 0; i < 7; i++)
    SendMessage(hwndCombo, CB_ADDSTRING, 0, (LPARAM)tc[i]);
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) {
    static HWND hwndCombo, hwndText;
    int iIndex, iLength, cxChar, cyChar;
    TCHAR *pVarName;
    switch (message) {
        case WM_CREATE:
            cxChar = LOWORD (GetDialogBaseUnits ());
            cyChar = HIWORD (GetDialogBaseUnits ());
            hwndCombo = CreateWindow (TEXT ("combobox"), NULL,
                        WS_CHILD | WS_VISIBLE | LBS_STANDARD, cxChar, cyChar * 3,
                        cxChar * 20 + GetSystemMetrics (SM_CXVSCROLL),
                        cyChar * 10, hwnd, (HMENU)ID_COMBO,
                        (HINSTANCE) GetWindowLong (hwnd, GWL_HINSTANCE), NULL);
            hwndText = CreateWindow (TEXT ("static"), NULL,
                        WS_CHILD | WS_VISIBLE | SS_LEFT, cxChar, cyChar,
                        GetSystemMetrics (SM_CXSCREEN), cyChar, hwnd,
                        (HMENU) ID_TEXT,
                        (HINSTANCE) GetWindowLong (hwnd, GWL_HINSTANCE), NULL);
            FillListBox (hwndCombo);
            return 0;
        case WM_SETFOCUS:
            SetFocus (hwndCombo);
            return 0;
        case WM_COMMAND:
            if (LOWORD(wParam) == ID_COMBO && HIWORD(wParam) == LBN_SELCHANGE) {
                iIndex = SendMessage(hwndCombo,CB_GETCURSEL,0,0);
                iLength = SendMessage(hwndCombo,CB_GETLBTEXTLEN,iIndex, 0)+1;
                pVarName = (char*) calloc (iLength, sizeof (TCHAR));
                SendMessage(hwndCombo, CB_GETLBTEXT, iIndex, (LPARAM)pVarName);
                SetWindowText(hwndText, pVarName);
                free(pVarName);
            }
            return 0;
        case WM_DESTROY:
            PostQuitMessage (0);
            return 0;
    }
    return DefWindowProc (hwnd, message, wParam, lParam);
}
```

H. LỚP SCROLL BAR

Trong các chương trước thanh cuộn cửa sổ được tạo ra bằng cách sử dụng hai thông số **WS_VSCROLL** hay **WS_HSCROLL**, thanh cuộn được tạo ra trong trường hợp này nằm ở bên dưới hoặc bên phải vùng client. Bây giờ tạo lập một control thanh cuộn mà kiểu điều khiển thanh cuộn này xuất hiện bất cứ nơi nào trong vùng thao tác. Bằng cách sử dụng lớp "**scrollbar**" cùng với hai kiểu thanh cuộn định nghĩa sẵn **SBS_VERT** và **SBS_HORZ**.

Không giống như các kiểu control điều kiển đã đề cặp ở trên, các control thanh cuộn ở đây không gởi thông điệp **WM_COMMAND**, mà lại gởi thông điệp **WS_VSCROLL** và **WS_HSCROLL** cho cửa sổ cha. Có thể phân biệt được thanh cuộn cửa sổ và kiểu điều khiển thanh cuộn thông qua tham số lParam. Tham số này sẽ bằng 0 nếu như thanh cuộn này là thanh cuộn cửa sổ, còn tham số này sẽ bằng handle của cửa sổ thanh

cuộn nếu như thanh cuộn đó là một kiểu điều khiển. Còn tham số wPram thì giống nhau cho cả hai loại thanh cuộn cửa sổ và kiểu điều khiển thanh cuộn.

Bạn có thể tạo một kiểu điều khiển thanh cuộn với chiều dài và chiều rộng tùy ý. Nếu bạn muốn tạo một kiểu điều khiển thanh cuộn có kích thước bằng kích thước của thanh cuộn cửa sổ. Thì dùng hai hàm sau để lấy chều cao và chiều rộng của thanh cuộn cửa sổ.

```
GetsystemMetrics (SM_CYHCROLL);
GetsystemMetrics (SM_CXVCROLL);
```

Tương tự như thanh cuộn cửa sổ chúng ta có thể đặt vùng và vị trí cho kiểu điều khiển thanh cuộn hàm.

```
SetScrollRange (hwndScroll, SB_CTL, iMin, iMax, bRedraw);
```

`hwnScroll` là handle của control thanh cuộn. Tham số **SB_CTL** là một trong hai kiểu **SBS_VERT** tương ứng với thanh cuộn ngang và **SBS_HORZ** tương ứng với thanh cuộn đứng. Theo mặc định thì thanh cuộn nằm trong vùng có giá trị từ 0 đến 100 đơn vị chiều dài, tuy nhiên có thể đặt lại vùng thanh cuộn thông qua hai tham số `iMin` tương ứng với chặn dưới của vùng và `iMax` tương ứng với chặn trên của vùng. Tham số `nRedraw` mang một trong hai giá trị `TRUE` hoặc `FALSE`, nếu muốn Windows vẽ lại thanh cuộn dựa trên vùng mới thì phải đặt giá trị này bằng `TRUE`, còn ngược lại thì đặt giá trị này bằng `FALSE`.

Dùng hàm sau để đặt lại vị trí con chạy trên thanh cuộn.

```
SetScrollPos (hwndScroll, SB_CTL, iPos, bRedraw);
```

Với `iPos` là vị trí cần đặt con chạy trên vùng control thanh cuộn.

1. Giao diện bàn phím đối với thanh cuộn

Có thể dùng bàn phím để di chuyển con chạy trên vùng control thanh cuộn khi thanh cuộn đó nhận được focus nhập. Sau đây là các phím di chuyển cùng với các chức năng của nó trên control thanh cuộn.

Các phím di chuyển	Giá trị wParam của thông điệp thanh cuộn
Home	SB_TOP
End	SB_BOTTOM
Page Up	SB_PAGEUP
Page Down	SB_PAGEDOWN
Left hay Up	SB_LINEUP
Right hay Down	SB_LINEDOWN

Bảng 3.4 Các giá trị wParam của thông điệp thanh cuộn

Đặt focus nhập cho các thanh cuộn bằng hàm.

```
SetFocus (hwndScroll);
```

Với `hwnScroll` là handle của control thanh cuộn. Muốn đặt focus cho một thanh cuộn nào đó khi khởi động chương trình thì phải đặt focus này khi xử lý thông điệp **WM_SETFOCUS** trong thủ tục **WndProc**. Vì lý do, thanh cuộn chỉ quan tâm đến các phím di chuyển, nó không quan tâm đến phím **Tab**. Điều này làm cho việc sử dụng phím **Tab** để di chuyển focus nhập từ thanh cuộn này đến thanh cuộn khác gặp nhiều khó khăn. Tuy nhiên để giải quyết vần đề này ta nguyên cứu các kỹ thuật sau.

Nhận địa chỉ thủ tục window của các control thanh cuộn bằng hàm **GetWindowLong** với định danh **GWL_ID**. Bạn có thể đặt thủ tục Windows cho thanh cuộn bằng hàm **SetWindowLong** với định danh **GWL_WNDPROC**. Trong ví dụ sau đây, hàm **SetWindownLong** đặt thủ tục window cho thanh cuộn mới và trả về địa chỉ thủ tục window thanh cuộn củ.

Ở hàm xử lý thanh cuộn **ScrollProc** trong ví dụ sau nhận tất cả các thông điệp gửi đến thủ tục window thanh cuộn. Hàm này chỉ đơn giản thay đổi **focus** nhập giữa các thanh cuộn khi bấm phím **Tab** hay **Shift-Tab**, bằng cách gọi window thanh cuộn củ thông qua hàm **CallWindowProc**.

2. Tô màu các thanh cuộn và các static text

Màu của các thanh cuộn được thực hiện bằng cách xử lý các thông điệp **WM_CTLCOLORSCROLLBAR**.

Trong ví dụ sau ở hàm **WinProc** ta định nghĩa một mảng gồm có 3 phần tử mỗi phần tử là một handle chổi tô (*brush*) cho một thanh cuộn.

```
static HBRUSH hBrush[3];
```

Các chổi tô trên được tạo ra bằng hàm **CreateSolidBrush** trong thủ tục xử lý thông điệp **WM_CREATE**.

```
for (i = 0; i < 3; i++) {
   hBrush [i] = CreateSolidBrush (color [i]);
}
```

Trong đó `color` chứa các giá trị màu sơ cấp **RGB** (*Red*, *Green*, *Blue*). Một trong 3 chổi tô (*brush*) được trả về khi xử lý thông điệp **WM_CTLCOLORSCROLLBAR** trong thủ tục **WinProc** bằng đoạn chương trình sau.

```
case WM_CTLCOLORSCROLLBAR:
   i = GetWindowLong ((HWND) lParam, GWL_ID);
   return (LRESULT) hBrush [i];
```

Chú ý: Các chổi tô phải được hủy bỏ trước khi kết thúc chương trình thông qua hàm **DeleteOject** khi xử lý thông điệp **WM_DESTROY** trong thủ tục xử lý **WinProc**.

```
for (i = 0; i < 3; i++)
   DeleteObject (hBrush [i]);
```

Các static text được tô màu tương tự như các thanh cuộn, bằng cách xử lý thông điệp **WM_CTLCOLORSTATIC** trong thủ tục **WinProc** thông qua hàm **SetTextColor**. Dùng hàm **SetBkColor** để đặt màu nền chữ, trong ví dụ sau thì màu nền chữ trùng với màu hệ thống **COLOR_BTNHIGHLIGHT**. Đối với các Static text tĩnh ta chỉ áp dụng màu nền chữ cho hình chữ nhật nằm sau mỗi ký tự chứ không phải toàn bộ chiều rộng của cửa sổ control. Để thực hiện điều này, trong thủ tục **WndProc** hàm xử lý **WM_CTLCOLORSTATIC** phải trả về một handle cho một chổi tô (brush) của màu **COLOR_BTNHIGHLIGHT**. Các chổi tô này được tạo ra khi xử lý thông điệp **WM_CREATE** và phải được hủy khi xử lý thông điệp **WM_DESTROY**.

3. Tô màu nền cửa sổ

Trong ví dụ dưới đây đã tạo ra cửa sổ và một chổi tô màu đen cho vùng thao tác cho cửa sổ này.

```
Wndclass.hbrBackground = CreateSolidBrush (0);
```

Khi thay đổi việc chọn trên các thanh cuộn thì chương trình tạo ra một chổi tô (*brush*) mới và chèn handle của chổi tô mới này vào cửa sổ ở trên. Nhận hay đặt handle của chổi tô (*brush*) bằng hàm **GetClassWord** và hàm **SetClassWord**. Có thể tạo một chổi tô (brush) mới và chèn handle của chổi tô này vào cấu trúc của cửa sổ, sau đó xóa chổi tô (*brush*) cũ.

```
DeleteObject ((HBRUSH)SetClassLong(hwnd, GCL_HBRBACKGROUND,
   (LONG)CreateSolidBrush(RGB (icolor[0], icolor[1], icolor[2]))));
```

Windows sẽ dùng chổi tô mới này để tô vùng thao tác cho lần tô kế tiếp. Dùng hàm sau để xóa 2/3 bên phải của vùng thao tác trước khi vẽ lại vùng này.

```
InvalidateRect (hwnd, &icolor, TRUE);
```

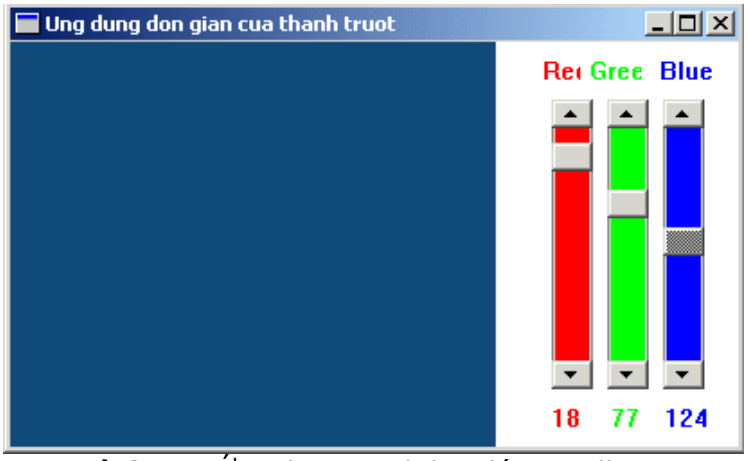
Hàm **InvalidateRect** làm cho window đặt thông điệp **WM_PAINT** vào trong hằng đợi các thông điệp của thủ tục xử lý window. Mà thông điệp **WM_PAINT** có độ ưu tiên thấp hơn các thông điệp xử lý bàn phím hay chuột nên nó được xử lý sau. Nếu muốn vùng cửa sổ được cập nhật lại màu ngay sau khi di chuyển con chạy trên thanh cuộn thì dùng hàm.

```
UpdateWindow (hwnd);
```

Tuy nhiên nếu chúng ta dùng hàm này thì làm cho tốc độ xử lý thông điệp chuột và bàn phím chậm lại.

Hàm **WndProc** không xử lý thông điệp **WM_PAINT** mà đưa cho hàm **DefWindowProc**. Như chúng ta biết windows xử lý mặc định thông điệp **WM_PAINT** chỉ đơn giản là lời gọi hàm **BeginPaint** và **EndPaint**. Nhưng ở trên chúng ta chỉ định **InvalidateRect** là phần nền cần phải xóa, chính điều này hàm **BeginPaint** sẽ khiến cho Windows tự động phát sinh thông điệp xóa nền **WM_ERASEBKGND**.

Chú ý: phải dọn dẹp các chổi tô trước khi chương trình kết thúc bằng hàm **DeleteObject** trong khi xử lý thông điệp **WM_DESTROY**.



Hình 3.4 Ứng dụng minh họa lớp Scroll Bar

4. Một ứng dụng của thanh cuộn

Hình 3.4 minh họa một ứng dụng về thanh cuộn (**Scroll Bar**). Trong ví dụ này, ta sử dụng 3 thanh cuộn và 6 static text. Bằng cách dùng chuột hay bàn phím, ta di chuyển con trượt trên thanh cuộn để thay đổi giá trị màu. Màu thay đổi tương ứng sẽ là màu tô của vùng client ở bên cạnh. Ba stactic text ở dưới mỗi thanh cuộn dùng để ghi nhận giá trị màu thay đổi tương ứng với các thanh cuộn.

SCROLLBAR.CPP *(trích dẫn)*

```
int idFocus;
WNDPROC OldScroll[3];
LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam) {
    static COLORREF crPrim[3] = {RGB (255, 0, 0), RGB (0, 255, 0),
                                 RGB (0, 0, 255)};
    static HBRUSH hBrush[3], hBrushStatic;
    static HWND hwndScroll[3], hwndLabel[3], hwndValue[3], hwndRect;
    static int color[3], cyChar;
    static RECT rcColor;
    static TCHAR *szColorLabel[] = { TEXT("Red"),
    TEXT("Green"), TEXT("Blue") };
    HINSTANCE hInstance;
    int i, cxClient, cyClient;
    TCHAR szBuffer[10];
    switch (message) {
        case WM_CREATE:
            hInstance = (HINSTANCE) GetWindowLong (hwnd, GWL_HINSTANCE);
            hwndRect = CreateWindow (TEXT("static"), NULL,
                        WS_CHILD | WS_VISIBLE | SS_WHITERECT, 0, 0, 0, 0, hwnd,
                        (HMENU) 9, hInstance, NULL);
            for (i = 0; i < 3; i++) {
                hwndScroll[i] = CreateWindow(TEXT ("scrollbar"), NULL,
                                 WS_CHILD | WS_VISIBLE | WS_TABSTOP | SBS_VERT,
                                 0, 0, 0, 0, hwnd, (HMENU) i, hInstance, NULL);
                SetScrollRange(hwndScroll[i], SB_CTL, 0, 255, FALSE);
                SetScrollPos (hwndScroll[i], SB_CTL, 0, FALSE);
                hwndLabel[i] = CreateWindow(TEXT("static"), szColorLabel[i],
                                 WS_CHILD|WS_VISIBLE|SS_CENTER, 0, 0, 0, 0, hwnd,
                                 (HMENU) (i + 3), hInstance, NULL);
                hwndValue[i] = CreateWindow(TEXT ("static"), TEXT ("0"),
                                 WS_CHILD | WS_VISIBLE | SS_CENTER, 0, 0, 0, 0,
                                 hwnd, (HMENU) (i + 6), hInstance, NULL);
                OldScroll[i] = (WNDPROC) SetWindowLong (hwndScroll[i], GWL_WNDPROC,
                                 (LONG) ScrollProc);
                hBrush[i] = CreateSolidBrush (crPrim[i]);
            }
            hBrushStatic = CreateSolidBrush (GetSysColor (COLOR_BTNHIGHLIGHT));
```

```
            cyChar = HIWORD (GetDialogBaseUnits ());
            return 0;
        case WM_SIZE:
            cxClient = LOWORD (lParam);
            cyClient = HIWORD (lParam);
            SetRect (&rcColor, 0, 0, cxClient/1.5, cyClient);
            MoveWindow (hwndRect, cxClient/1.5, 0, cxClient/2, cyClient, TRUE);
            for (i = 0; i < 3; i++) {
                MoveWindow(hwndScroll[i],
                    ((int)(1.2 * i + 1) * cxClient) / 13 + (int)cxClient/1.5,
                    2*cyChar, cxClient / 17, cyClient - 4 * cyChar, TRUE);
                MoveWindow(hwndLabel[i],
                    (2.3 * i + 1) * cxClient / 28+ cxClient/1.5, cyChar / 2,
                    cxClient / 8, cyChar, TRUE);
                MoveWindow (hwndValue[i],
                    (2.3 * i + 1) * cxClient / 28+ cxClient/1.5,
                    cyClient - 3 * cyChar / 2, cxClient / 8, cyChar, TRUE);
            }
            SetFocus (hwnd);
            return 0;
        case WM_SETFOCUS:
            SetFocus (hwndScroll[idFocus]);
            return 0;
        case WM_VSCROLL:
            i = GetWindowLong ((HWND) lParam, GWL_ID);
            switch (LOWORD (wParam)) {
                case SB_PAGEDOWN:
                    color[i] += 15;
                case SB_LINEDOWN:
                    color[i] = min (255, color[i] + 1);
                    break;
                case SB_PAGEUP:
                    color[i] -= 15;
                case SB_LINEUP:
                    color[i] = max (0, color[i] - 1);
                    break;
                case SB_TOP:
                    color[i] = 0;
                    break;
                case SB_BOTTOM:
                    color[i] = 255;
                    break;
                case SB_THUMBPOSITION:
                case SB_THUMBTRACK:
                    color[i] = HIWORD (wParam);
                    break;
                default:
                    break;
            }
            SetScrollPos (hwndScroll[i], SB CTL, color[i], TRUE);
            wsprintf (szBuffer, TEXT ("%i"), color[i]);
            SetWindowText(hwndValue[i], szBuffer);
            DeleteObject((HBRUSH)SetClassLong(hwnd, GCL_HBRBACKGROUND,
                 (LONG)CreateSolidBrush(RGB(color[0], color[1], color[2]))));
            InvalidateRect(hwnd, &rcColor, TRUE);
            return 0;
        case WM_CTLCOLORSCROLLBAR:
            i = GetWindowLong((HWND) lParam, GWL_ID);
            return (LRESULT) hBrush[i];
```

```
        case WM_CTLCOLORSTATIC:
            i = GetWindowLong ((HWND) lParam, GWL_ID);
            if (i >= 3 && i <= 8) {
                SetTextColor((HDC) wParam, crPrim[i % 3]);
                SetBkColor((HDC) wParam, GetSysColor (COLOR_BTNHIGHLIGHT));
                return (LRESULT) hBrushStatic;
            }
            break;
        case WM_SYSCOLORCHANGE:
            DeleteObject(hBrushStatic);
            hBrushStatic = CreateSolidBrush (GetSysColor (COLOR_BTNHIGHLIGHT));
            return 0;
        case WM_DESTROY:
            DeleteObject((HBRUSH) SetClassLong (hwnd, GCL_HBRBACKGROUND,
                   (LONG) GetStockObject (WHITE_BRUSH)));
            for (i = 0; i < 3; i++)
                DeleteObject(hBrush[i]);
            DeleteObject(hBrushStatic);
            PostQuitMessage (0);
            return 0;
    }
    return DefWindowProc (hwnd, message, wParam, lParam);
}

LRESULT CALLBACK ScrollProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    int id = GetWindowLong (hwnd, GWL_ID);
    switch (message) {
        case WM_KEYDOWN:
            if (wParam == VK_TAB)
                SetFocus(GetDlgItem(GetParent(hwnd),
                    (id + (GetKeyState(VK_SHIFT) < 0 ? 2: 1)) % 3));
            break;
        case WM_SETFOCUS:
            idFocus = id;
            break;
    }
    return CallWindowProc(OldScroll[id],hwnd,message,wParam,lParam);
}
```

CHƯƠNG IV: XỬ LÝ BÀN PHÍM, THIẾT BỊ CHUỘT, VÀ BỘ ĐỊNH THỜI GIAN

A. MỞ ĐẦU

Các chương trước đã trình bày các thành phần điều khiển chung và hộp thoại. Trong các thành phần này, việc giao tiếp với người dùng đã được các hộp thoại hay các điều khiển xử lý, thường không cần quan tâm lắm việc giao tiếp với các thiết bị đó. Tuy nhiên, lập trình trên Windows cũng cần phải hiểu việc xử lý các thiết bị nhập như bàn phím và thiết bị chuột. Một số ứng dụng như đồ họa hay thao tác văn bản ít nhiều cũng phải viết các xử lý liên quan tới bàn phím và thiết bị chuột.

Trong chương này, hai phần đầu sẽ trình bày cách người lập trình sử dụng bàn phím và thiết bị chuột để xây dựng một ứng dụng trên Windows. Thực chất của việc xử lý bàn phím hay thiết bị chuột cũng đơn giản, vì với cơ chế thông điệp của Windows thì ta chỉ cần tìm hiểu các thông điệp được phát sinh từ bàn phím hay từ thiết bị chuột để viết các xử lý tương ứng với từng thiết bị.

Phần cuối của chương trình bày một thành phần cũng không kém quan trọng là bộ định thời gian. Windows cung cấp cơ chế này để truyền thông với ứng dụng theo định kì. Với cơ chế này, ứng dụng chỉ cần khai báo một bộ định thời gian với một khoảng thời gian cho trước. Và khi ứng dụng hoạt động thì hệ thống sẽ truyền một tín hiệu cho ứng dụng theo từng khoảng thời gian định kì đã được khai báo.

Tóm lại việc tìm hiểu bàn phím, thiết bị chuột, và bộ định thời gian sẽ đem lại sự hiểu biết sâu sắc về thành phần nhập liệu căn bản của một ứng dụng trên Windows.

B. BÀN PHÍM

Trong môi trường tương tác đồ hoạ ngày nay có hai thành phần nhập liệu không thể thiếu là bàn phím (*keyboard*) và thiết bị chuột (*mouse*). Tuy một số ứng dụng không tiện lợi khi dùng bàn phím, như các chương trình trò chơi (*game*), hay các mô phỏng đồ họa với thiết bị định vị là thiết bị chuột chẳng hạn, nhưng bàn phím vẫn là thiết bị không thể thay thế của một máy tính. Bàn phím hỗ trợ nhập liệu rất phong phú: một chương trình soạn thảo văn bản thì không thể thiếu việc nhập dữ liệu từ bàn phím. Nếu máy tính bị hư thiết bị chuột bị ta vẫn có thể thực thi các tác vụ của ứng dụng một cách bình thường.

Trong phần này chúng ta sẽ tìm hiểu các thông điệp được phát sinh từ bàn phím và cách can thiệp để xử lý chúng.

1. Nền tảng cơ sở về bàn phím

Windows nhận và xử lý thông tin nhận được từ bàn phím, qua hình thức các thông điệp và gởi cho ứng dụng. Trong ứng dụng Windows các thông điệp sẽ được hệ điều hành chuyển cho hàm xử lý cửa sổ **`WndProc`** của ứng dụng.

Windows cung cấp 8 loại thông điệp khác nhau để phân biệt các tình huống của các phím được gõ. Tuy nhiên không phải lúc nào chúng ta cũng phải xử lý toàn bộ các thông điệp đó, thông thường thì chỉ cần xử lý một nửa các thông điệp được phát sinh từ bàn phím, và các thông điệp còn lại sẽ được Windows xử lý mặc định.

Ví dụ, trong Windows có thể bỏ qua khi nhấn phím **Ctrl**, **Alt**, **Shift** cùng với các phím khác, nếu không muốn chặn để xử lý riêng cho ứng dụng. Trong trường hợp nếu chúng ta muốn chặn để xử lý riêng cho mình, chẳng hạn tạo phím nóng (*hotkey*) phải chú ý tránh dùng trùng hợp với các phím nóng mà Windows cung cấp. Vì khi đó theo quyền ưu tiên, ứng dụng của chúng ta sẽ xử lý thông điệp đó và sẽ làm cho hệ thống không hoạt động như bình thường.

Thành phần giao tiếp chung như hộp thoại (*dialog*) cũng có nhiều giao tiếp với bàn phím, nhưng chúng ta không cần quan tâm đến việc giao tiếp với bàn phím khi hộp thoại được kích hoạt. Xử lý bàn phím trong hộp thoại thường được giao cho Windows xử lý. Các điều khiển được dùng trong hộp thoại như hộp nhập liệu (*edit box*), hộp lựa chọn (*check box*), hay các nút nhấn (*button*)...điều có khả năng tự xử lý phím gõ vào và chỉ trả lại thông báo của các phím gõ cho cửa sổ cha (*parent window*). Tuy vậy, với một số các điều khiển nhất định theo ứng dụng ta có thể phải xử lý các thông điệp để tăng cường thêm sức mạnh của thành phần điều khiển này.

Tóm lại trong các ứng dụng được cấu thành từ các thành phần điều khiển cơ bản này thì chúng ta không quan tâm đến việc xử lý các thông điệp từ bàn phím.

2. Khái niệm focus trong các ứng dụng trong môi trường Windows

Windows đưa ra khái niệm sự quan tâm (*focus*) cho các ứng dụng được chạy đồng thời trong một thời điểm. Vì chỉ có một bàn phím, nên Windows phải quản lý và phân phối các thông điệp được gõ vào cho các ứng dụng. Thông thường, có các trường hợp trên Windows là: một trong số các ứng dụng đang chạy được

kích hoạt (*active*) hay không có ứng dụng nào chạy. Khi có một chương trình ứng dụng được kích hoạt thì Windows xem như ứng dụng đó nhận được sự quan tâm.Trong một ứng dụng có nhiều các cửa sổ, mỗi thời điểm chỉ một cửa sổ nhận được sự quan tâm. Theo cơ chế này, Windows cung cấp một dạng gọi là hàng đợi thông điệp. Mỗi thông điệp sẽ được đưa vào hàng đợi xử lý thông điệp và được Windows phân phối đến các ứng dụng tương ứng.

Hàm **DispatchMessage** trong vòng lặp xử lý thông điệp sẽ chịu trách nhiệm chuyển thông điệp đến thủ tục xử lý cửa sổ **WndProc** của các cửa sổ tương ứng.

Một cửa sổ có thể xác định được trạng thái quan tâm của mình bằng cách chặn các thông điệp **WM_SETFOCUS**, **WM_KILLFOCUS** trong hàm xử lý **WndProc**. Thông điệp **WM_SETFOCUS** sẽ cho cửa sổ biết được thời điểm nhận được quan tâm của Windows và ngược lại **WM_KILLFOCUS** sẽ thông báo cho cửa sổ biết được đã mất sự quan tâm từ Windows. Phần sau sẽ giới thiệu kỹ hơn về xử lý thông điệp.

3. Cơ chế hàng đợi và quản lý hàng đợi

Trong Windows khi người dùng nhấn và nhả phím trên bàn phím, thì thông qua trình điều khiển thiết bị bàn phím (*keyboard driver*) sẽ diễn dịch mã quét (*scan code*) của phần cứng sang hình thức thông điệp. Trước hết Windows sẽ tạm thời lưu trữ thông điệp này vào hàng đợi thông điệp của hệ thống (*system message queue*). Hàng đợi thông điệp hệ thống của Windows là một hàng đợi duy nhất và quản lý các thao tác tiền xử lý thông tin nhập từ bàn phím và chuột. Windows sẽ lần lượt lấy các thông điệp trong hàng đợi xử lý và sẽ gởi đến hàng đợi của ứng dụng khi ứng dụng đã xử lý xong thông điệp bàn phím và thiết bị chuột trước đó.

Lý do mà Windows phải chia thành hai giai đoạn trong quá trình nhận và gởi thông điệp từ bàn phím đến hàng đợi của ứng dụng là do việc đồng bộ hóa với mọi tiến trình. Nếu Windows không quản lý hàng đợi hệ thống thì rất khó đồng bộ các tiến trình của các ứng dụng. Ví dụ, khi một cửa sổ nhận được sự quan tâm và chuẩn bị xử lý các thông điệp. Người dùng có thể gõ phím nhanh trong khi thông điệp trước vẫn chưa xử lý xong. Giả sử người dùng muốn chuyển qua ứng dụng khác và nhấn **Alt-Tab**, khi đó thông điệp bàn phím mới này sẽ được đưa vào hàng đợi của hệ thống và phân phát cho ứng dụng kia chứ không phải đưa vào hàng đợi của ứng dụng. Với tính năng đồng bộ hóa của Windows thì các thông điệp từ bàn phím đảm bảo được chuyển giao đúng cho các cửa sổ tương ứng.

4. Xử lý thông điệp từ bàn phím

a. Thông điệp từ thao tác nhấn và nhả của một phím

Thao tác căn bản của nhập liệu từ bàn phím là thao tác nhấn và nhả một phím. Khi thao tác nhấn một phím trên bàn phím được thực hiện thì Windows sẽ phát sinh thông điệp **WM_KEYDOWN** hay **WM_SYSKEYDOWN** và đưa vào hàng đợi thông điệp của ứng dụng hay cửa sổ nhận được sự quan tâm (*focus*). Cũng tương tự, khi ta nhả phím thì thông điệp **WM_KEYUP** hay **WM_SYSKEYUP** sẽ được hệ điều hành phân phát tới hàng đợi đó.

Khi thực hiện thao tác nhập từ bàn phím thì việc nhấn (*down*) và nhả (*up*) phải đi đôi với nhau. Tuy nhiên, nếu chúng ta nhấn một phím và giữ luôn thì Windows sẽ phát sinh hàng loạt các thông điệp **WM_KEYDOWN** hay **WM_SYSKEYDOWN**, nhưng với thao tác nhả thì chỉ phát sinh một thông điệp **WM_KEYUP** hay **WM_SYSKEYUP**. Các thông điệp này sẽ được gởi tuần tự đến các hàm WndProc của các cử sổ nhận được sự quan tâm của Windows. Ngoài ra chúng ta có thể biết được thời điểm mà thông điệp phím được gõ vào lúc nào bằng cách dùng hàm **GetMessageTime**.

Như chúng ta đã thấy trong dòng trên một thao tác nhấn hay thả thì có hai dạng thông điệp khác nhau như **WM_KEYDOWN** và **WM_SYSKEYDOWN** của thao tác nhấn, và tương tự đối với thao tác nhả. Thông điệp có tiếp đầu ngữ là "**SYS**" thường được phát sinh khi người dùng nhấn các phím gõ hệ thống. Khi người dùng nhấn phím **Alt** kết hợp với phím khác thì thường phát sinh thông điệp **WM_SYSKEYDOWN** và **WM_SYSKEYUP**. Đối với tổ hợp các phím Alt, chức năng thường là gọi một mục chọn trên trình đơn menu của ứng dụng hay trình đơn hệ thống system menu và ngoài ra dùng để chuyển đổi các tác vụ giữa nhiều ứng dụng khác nhau (phím **Alt+Tab** hay **Alt+Esc**), và đóng cửa sổ ứng dụng khi kết hợp với phím chức năng **F4**.

Khi xây dựng một chương trình ứng dụng, thường ít quan tâm đến các thông điệp **WM_SYSKEYDOWN** và thông điệp **WM_SYSKEYUP**. Chúng ta chỉ cần dùng hàm **DefWindowProc** cuối mỗi hàm **WndProc** của cửa sổ nhận thông điệp. Windows sẽ chịu trách nhiệm xử lý các thông điệp dạng hệ thống này, nếu có những tác vụ đặc biệt thì chúng ta có thể chặn các thông điệp này để xử lý. Tuy nhiên ta không nên làm như vậy vì khi đó chương trình của chúng ta sẽ chạy không bình thường như các ứng dụng khác. Không phải chúng ta giao phó hoàn toàn cho Windows xử lý các thông điệp hệ thống của ứng dụng, mà Windows sẽ xử lý các thông điệp hệ thống này và đưa ra các thông điệp bình thường khác đến ứng dụng. Ví dụ khi nhấn phím **Alt+Tab** thì Windows sẽ tạo một thông điệp hệ thống gởi vào hàng đợi của ứng dụng và khi không xử lý

thông điệp này thì theo mặc định hàm **DefWindowProc** sẽ xử lý và trả về thông điệp **WM_KILLFOCUS** cho ứng dụng, khi đó ứng dụng của chúng ta sẽ dễ dàng xử lý hơn.

Tóm lại thông điệp **WM_KEYDOWN** và **WM_KEYUP** thường được sinh ra bởi các phím nhấn thông thường không kết hợp với phím **Alt**. Nếu chương trình của chúng ta bỏ qua không xử lý các thông điệp này thì Windows cũng không tạo ra các thông điệp hay xử lý gì đặc biệt.

b. Các thông điệp phát sinh từ bàn phím

Sau đây là bảng mô tả các thông điệp phát sinh từ bàn phím (theo thứ tự Alphabet).

Thông điệp	Nguyên nhân phát sinh
WM_ACTIVATE	Thông điệp này cùng được gởi đến các cửa sổ bị kích hoạt và cửa sổ không bị kích hoạt. Nếu các cửa sổ này cùng một hàng đợi nhập liệu, các thông điệp này sẽ được truyền một cách đồng bộ, đầu tiên thủ tục Windows của cửa sổ trên cùng bị mất kích hoạt, sau đó đến thủ tục của cửa sổ trên cùng được kích hoạt. Nếu các cửa sổ này không nằm trong cùng một hàng đợi thì thông điệp sẽ được gởi một cách không đồng bộ, do đó cửa sổ sẽ được kích hoạt ngay lập tức.
WM_APPCOMMAND	Thông báo đến cửa sổ rằng người dùng đã tạo một sự kiện lệnh ứng dụng, ví dụ khi người dùng kích vào button sử dụng chuột hay đánh vào một kí tự kích hoạt một lệnh của ứng dụng.
WM_CHAR	Thông điệp này được gởi tới cửa sổ có sự quan tâm khi thông điệp WM_KEYDOWN đã được dịch từ hàm **TranslateMessage**. Thông điệp WM_CHAR có chứa mã kí tự của phím được nhấn.
WM_DEADCHAR	Thông điệp này được gởi tới cửa sổ có sự quan tâm khi thông điệp WM_KEYUP đã được xử lý từ hàm **TranslateMessage**. Thông điệp này xác nhận mã kí tự khi một phím dead key được nhấn. Phím dead key là phím kết hợp để tạo ra kí tự ngôn ngữ không có trong tiếng anh (xuất hiện trong bàn phím hỗ trợ ngôn ngữ khác tiếng Anh).
WM_GETHOTKEY	Ứng dụng gởi thông điệp này để xác định một phím nóng liên quan đến một cửa sổ. Để gởi thông điệp này thì dùng hàm **SendMessage**.
WM_HOTKEY	Thông điệp này được gởi khi người dùng nhấn một phím nóng được đăng kí trong **RegisterHotKey**.
WM_KEYDOWN	Thông điệp này được gởi cho cửa sổ nhận được sự quan tâm khi người dùng nhấn một phím trên bàn phím. Phím này không phải phím hệ thống (Phím không có nhấn phím **Alt**).
WM_KEYUP	Thông điệp này được gởi cho cửa sổ nhận được sự quan tâm khi người dùng nhả một phím đã được nhấn trước đó.Phím này không phải phím hệ thống (Phím không có nhấn phím **Alt**).
WM_KILLFOCUS	Thông điệp này được gởi tới cửa sổ đang nhận được sự quan tâm trước khi nó mất quyền này.
WM_SETFOCUS	Thông điệp này được gởi tới cửa sổ sau khi cửa sổ nhận được sự quan tâm của Windows
WM_SETHOTKEY	Ứng dụng sẽ gởi thông điệp này đến cửa sổ liên quan đến phím nóng, khi người dùng nhấn một phím nóng thì cửa sổ tương ứng liên quan tới phím nóng này sẽ được kích hoạt.
WM_SYSCHAR	Thông điệp này sẽ được gởi tới cửa sổ nhận được sự quan tâm khi hàm **TranslateMesage** xử lý xong thông điệp WM_SYSKEYDOWN. Thông điệp WM_SYSCHAR chứa mã cửa phím hệ thống. Phím hệ thống là phím có chứa phím **Alt** và tổ hợp phím khác.
WM_SYSDEADCHAR	Thông điệp này được gởi tới cửa sổ nhận được sự quan tâm khi một thông điệp **WM_SYSKEYDOWN** được biên dịch trong hàm **TranslateMessage**. Thông điệp này xác nhận mã kí tự của phím hệ thống deadkey được nhấn.
WM_SYSKEYDOWN	Thông điệp này được gởi tới cửa sổ nhận được sự quan tâm khi người dùng nhấn phím **F10** hay nhấn **Alt** trước khi nhấn phím khác. Thông điệp này cũng được gởi khi không có cửa sổ nào nhận được sự quan tâm và lúc này thì cửa sổ nhận được là cửa sổ đang được kích hoạt (*Active*).

WM_SYSKEYUP	Thông điệp này được gởi tới cửa sổ nhận được sự quan tâm khi người dùng nhấn một phím mà trước đó đã giữ phím **Alt**. Cũng tương tự nếu không có cửa sổ nào nhận được sự quan tâm thì thông điệp này sẽ được gởi cho cửa sổ đang được kích hoạt.

Bảng 4.1 Mô tả thông điệp phát sinh từ bàn phím

c. Mã phím ảo (Virtual key code)

Windows cung cấp khái niệm phím ảo nhằm tách rời với thiết bị bàn phím hay nói cách khác là tiến tới độc lập thiết bị với bàn phím. Khi một phím được nhấn thì phần cứng vật lý phát sinh ra một mã quét (*scan code*), trên bàn phím tương thích IBM các phím được gán với các mã ví dụ phím **W** là **17**, phím **E** là **18**, và phím **R** là **19**... Cách sắp xếp này thuần túy dựa trên vị trí vật lý của phím trên bàn phím. Những người xây dựng nên Windows nhận thấy rằng nếu dùng trực tiếp mã quét thì sẽ không thích hợp khi bị lệ thuộc vào bàn phím hiện tại và tương lai. Do đó họ cố xử lý bàn phím bằng cách độc lập thiết bị hơn, bằng cách tạo ra một bảng định nghĩa tập giá trị phím tổng quát mà sau này được gọi là mã phím ảo. Một số giá trị bàn phím ảo mà ta không thấy xuất hiện trên bàn phím IBM tương thích nhưng có thể tìm thấy chúng trong bàn phím của các nhà sản xuất khác hay chúng được để dành cho bàn phím trong tương lai. Các giá trị của phím ảo này được định nghĩa trong tập tin tiêu đề **WINUSER.H** và được bắt đầu với các tiếp đầu ngữ **VK_xxxxx**. sau đây là bảng mô tả các phím ảo thông dụng trong Windows giao tiếp với bàn phím IBM tương thích.

Thập phân	Thập lục phân	Hằng phím định nghĩa trong WINUSER.H	Windows dùng	Bàn phím IBM tương thích
1	01	VK_LBUTTON		Nút chuột trái
2	02	VK_RBUTTON		Nút chuột phải
3	03	VK_CANCEL	X	**Ctrl –Break**
4	04	VK_MBUTTON		Nút chuột giữa

Bảng 4.2 Mô tả các phím ảo

Các thông điệp trên chỉ được nhận khi dùng chuột, ta không bao giờ nhận được thông điệp trên nếu gõ từ bàn phím. Không nên dùng phím **Ctrl–Break** trong ứng dụng Windows, phím này thường được ngắt các ứng dụng trong DOS.

Những giá trị phím ảo tiếp sai dành cho các phím **Backspace**, **Tab**, **Enter**, **Escape**, và **Spacebar** được dùng nhiều trong chương trình Windows.

Thập phân	Thập lục phân	Hằng phím định nghĩa trong WINUSER.H	Windows dùng	Bàn phím IBM tương thích
8	08	VK_BACK	X	**Backspace**
9	09	VK_TAB	X	**Tab**
12	0C	VK_CLEAR	X	Phím số 5 trong **NumPad** (**Numlock** tắt)
13	0D	VK_RETURN	X	**Enter** (cho hai phím)
16	10	VK_SHIFT	X	**Shift** (cho hai phím)
17	11	VK_CONTROL	X	**Ctrl** (cho hai phím)
18	12	VK_MENU	X	**Alt** (cho hai phím)
19	13	VK_PAUSE	X	Pause
20	14	VK_CAPITAL	X	Caps Lock
27	1B	VK_ESCAPE	X	Esc
32	20	VK_SPACE	X	Spacebar

Bảng 4.3 Mô tả các phím ảo

Bảng mô tả phím ảo tiếp sau đây là các phím thường được sử dụng nhiều trong Windows.

Thập phân	Thập lục phân	Hằng phím định nghĩa trong WINUSER.H	Windows dùng	Bàn phím IBM tương thích
33	21	VK_PRIOR	X	**Page Up**
34	22	VK_NEXT	X	**Page Down**
35	23	VK_END	X	**End**
36	24	VK_HOME	X	**Home**

37	25	VK_LEFT	X	Phím trái
38	26	VK_UP	X	Phím mũi tên lên
39	27	VK_RIGHT	X	Phím phải
40	28	VK_DOWN	X	Phím xuống
41	29	VK_SELECT		-
42	2A	VK_PRINT		-
43	2B	VK_EXECUTE		-
44	2C	VK_SNAPSHOT		**Print Screen**
45	2D	VK_INSERT	X	**Insert**
46	2E	VK_DELETE	X	**Delete**
47	2F	VK_HELP		

Bảng 4.4 Mô tả các phím ảo (tiếp theo)

Một số các phím ảo như **VK_SELECT**, **VK_PRINT**, **VK_EXECUTE**, hay **VK_HELP** thường chỉ xuất hiện trong các bàn phím giả lập mà chúng ta ít khi nhìn thấy.

Tiếp sau là mã phím ảo của các phím số và phím chữ trên bàn phím chính. Trong bàn phím bổ sung phím **Num Pad** được qui định riêng.

Thập phân	Thập lục phân	Hằng phím định nghĩa trong WINUSER.H	Windows dùng	Bàn phím IBM tương thích
48-57	30-39	Không	X	Từ phím **0** đến phím **9** trên bàn phím chính
65-90	41-5A	Không	X	Từ phím **A** đến phím **Z**.

Bảng 4.5 Mô tả các phím chữ và số trên bàn phím chính

Mã phím ảo trong bảng trên bằng với mã ASCII của kí tự mà phím thể hiện. Trong các ứng dụng Windows thường không dùng mã phím ảo này mà thay vào đó ứng dụng chỉ xử lý thông điệp kí tự dành cho kí tự có mã ASCII.

Cuối cùng là mã phím ảo của bàn phím mở rộng **Num Pad** (bên phải của bàn phím) và các phím chức năng (**F1**, **F2**, **F3**...).

Thập phân	Thập lục phân	Hằng phím định nghĩa trong WINUSER.H	Windows dùng	Bàn phím IBM tương thích
96-105	60-69	VK_NUMPAD0 đến VK_NUMPAD9		Phím **0 – 9** khi đèn **Num Lock** được bật
106	6A	VK_MULTIPLY		Phím *****
107	6B	VK_ADD		Phím **+**
108	6C	VK_SEPARATOR		
109	6D	VK_SUBTRACT		Phím **-**
110	6E	VK_DECIMAL		Phím.
111	6F	VK_DIVIDE		Phím **/**
112-121	70-79	VK_F1 đến VK_F10	X	Phím **F1** đến **F10**
122-135	7A-87	VK_F11 đến VK_F24		Phím **F11** đến **F24**
144	90	VK_NUMLOCK		**Num Lock**
145	91	VK_SCROLL		**Scroll Lock**

Bảng 4.6 Mô tả các phím bổ sung

Một số các phím ảo như **F13**- **F24** thì được Windows tạo ra phòng hờ cho các bàn phím sau này. Thường thì ứng dụng Windows chỉ dùng phím **F1** – **F10** mà thôi.

d. Các tham số wParam và lParam trong thông điệp phát sinh từ bàn phím

Trong thông điệp phím gõ (WM_KEYDOWN, WM_KEYUP, WM_SYSKEYDOWN, và WM_SYSKEYUP) thì tham số **wParam** sẽ nhận giá trị mã phím ảo. Còn tham số **lParam** sẽ chứa thông tin chi tiết về phím được gõ vào.

Bảng sau mô tả các thông tin chứa trong 32 bit của tham số lParam.

Bit nhận dạng	Nội dung

0-15	Chứa số lần của phím được nhấn xuống. Nếu chúng ta chỉ nhấn rồi nhả ra thì giá trị này bằng 1, nếu chúng ta giữ luôn phím thì số lập này sẽ tăng theo.
16-23	Chứa mã quét **OEM** (*Original Equipment Manufacturer*) được phát sinh từ phần cứng, ta hầu như không quan tâm đến thông tin này
24	Cờ này có giá trị 1 khi phím được nhấn là phím thuộc nhóm phím mở rộng trên bàn phím IBM tương thích hay phím **Alt**, **Ctrl** bên phải bàn phím được nhấn. Windows ít quan tâm đến giá trị của cờ này
29	Mã ngữ cảnh của phím (*Context code*), nếu cờ này có giá trị bằng 1 thì phím **Alt** được nhấn, điều này tương ứng với thông điệp **WM_SYSKEYDOWN** hay **WM_SYSKEYUP** được phát sinh
30	Trạng thái của phím trước đó, trường này bằng 0 cho biết phím trước đó ở trạng thái nhả, và 1 nếu phím trước đó ở trạng thái nhấn.
31	Trạng thái dịch chuyển của phím, cờ này bằng 0 nếu phím đang được nhấn, và bằng 1 nếu phím được nhấn.

Bảng 4.7 Mô tả thông tin chứa trong tham số lParam

e. Đoạn chương trình minh họa

Đoạn chương trình dưới đây minh họa một chương trình nhỏ nhập các kí tự từ bàn phím và xuất ra màn hình.

```
#define BUFSIZE 65535;
#define SHIFTED 0x8000;
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam) {
    HDC hdc; // Thiết bị ngữ cảnh
    TEXTMETRIC tm; // Cấu trúc metric của văn bản
    static DWORD dwCharX; // Bề ngang của kí tự
    static DWORD dwCharY; // Chiều dài c?a kí tự
    static DWORD dwClientX; // Bề ngang của vùng làm việc
    static DWORD dwClientY; // Chiều dài của vùng làm việc
    static DWORD dwLineLen; // Chiều dài của một dòng
    static DWORD dwLines; // Số dòng văn bản trong vùng làm việc
    static int nCaretPosX = 0; // Tọa độ x của caret
    static int nCaretPosY = 0; // Tọa độ y của caret
    static int nCharWidth = 0; // Bề dày của kí tự
    static int cch = 0; // Số kí tự trong buffer
    static int nCurChar = 0; // Chỉ đến kí tự hiện thời trong buffer
    static PTCHAR pchInputBuf; // Kí tự tìm đưa vào buffer
    int i, j; // Biến lặp
    int cCR = 0; // Số kí tự xuống dòng
    int nCRIndex = 0; // Chỉ đến kí tự xuống dòng cuối cùng
    int nVirtKey; // Mã phím ảo
    TCHAR szBuf [128]; // Buffer tạm
    TCHAR ch; // Kí tự
    PAINTSTRUCT ps; // Dùng cho hàm BeginPaint
    RECT rc; // Hình chữ nhật trong hàm DrawText
    SIZE sz; // Kích thước của chuỗi
    COLORREF crPrevText; // Màu của văn bản
    COLORREF crPrevBk; // Màu nền
    switch (message) {
        case WM_CREATE:
            /* Lấy thông tin font hiện thời */
            hdc = GetDC (hWnd);
            GetTextMetrics (hdc, &tm);
            ReleaseDC (hWnd, hdc);
            dwCharX = tm.tmAveCharWidth;
            dwCharY = tm.tmHeight;
            /* Cấp phát bộ nhớ để lưu kí tự nhập vào */
            pchInputBuf = (LPTSTR)GlobalAlloc(GPTR,BUFSIZE * sizeof (TCHAR));
            return 0;
        case WM_SIZE:
```

```
            /* Luu giữ kích thước của vùng làm việc */
            dwClientX = LOWORD (lParam);
            dwClientY = HIWORD (lParam);
            /* Tính kích thước tối đa của một dòng
               và số dòng tối da trong vùng làm việc */
            dwLineLen = dwClientX / dwCharX;
            dwLines = dwClientY / dwCharY;
            break;
        case WM_SETFOCUS:
            /* Tạo và hiển thị caret khi cửa sổ focus */
            CreateCaret (hWnd, (HBITMAP) 1, 0, dwCharY);
            SetCaretPos (nCaretPosX, nCaretPosY * dwCharY);
            ShowCaret (hWnd);
            break;
        case WM_KILLFOCUS:
            /* Ẩn caret và hủy khi cửa sổ mất focus */
            HideCaret (hWnd);
            DestroyCaret ();
            break;
        case WM_CHAR:
            switch (wParam) {
                case 0x08: // Backspace
                case 0x0A: // Linefeed
                case 0x1B: // Escape
                    MessageBeep((UINT) -1);
                    return 0;
                case 0x09: // tab
                    /* Chuyển phím tab thành 4 kí tự trống liên tục nhau */
                    for (i = 0; i < 4; i ++)
                        SendMessage (hWnd, WM_CHAR, 0x20, 0);
                        return 0;
                case 0x0D: // Xuống dòng
                    /* Lưu kí tự xuống dòng và tạo dòng mới đưa caret
                       xuống vị trí mới */
                    pchInputBuf [cch++] = 0x0D;
                    nCaretPosX = 0;
                    nCaretPosY += 1;
                    break;
                default: // Xử lý nhung kí tự có thể hiển thị được
                    ch = (TCHAR) wParam;
                    HideCaret (hWnd);
                    /* Lấy bề dày của kí tự và xuất ra */
                    hdc = GetDC (hWnd);
                    GetCharWidth32 (hdc, (UINT) wParam, (UINT)wParam, &nCharWidth);
                    TextOut(hdc, nCaretPosX, nCaretPosY*dwCharY, &ch, 1);
                    ReleaseDC (hWnd, hdc);
                    /* Lưu kí tự vào buffer */
                    pchInputBuf [cch++] = ch;
                    /* Tính lại vị trí ngang của caret. Nếu vị trí này là cuối dòng
                       thì chèn thêm kí tự xuống dòng và di chuyển caret đến dòng
                       mới */
                    nCaretPosX += nCharWidth;
                    if ((DWORD) nCaretPosX > dwLineLen) {
                        nCaretPosX = 0;
                        pchInputBuf [ cch++ ] = 0x0D;
                        ++nCaretPosY;
                    }
                    nCurChar = cch;
                    ShowCaret (hWnd);
```

```
                break;
            }
            SetCaretPos (nCaretPosX, nCaretPosY * dwCharY);
            break;
        case WM_KEYDOWN:
            switch (wParam) {
                case VK_LEFT: // LEFT arrow
                    /* Caret di chuyển qua trái và chỉ đến đầu dòng */
                    if (nCaretPosX > 0) {
                        HideCaret (hWnd);
                        /* Tính toán lại vị trí của caret khi qua trái */
                        ch = pchInputBuf [--nCurChar ];
                        hdc = GetDC (hWnd);
                        GetCharWidth32 (hdc, ch, ch, &nCharWidth);
                        ReleaseDC (hWnd, hdc);
                        nCaretPosX=max(nCaretPosX-nCharWidth,0);
                        ShowCaret (hWnd);
                    }
                    break;
                case VK_RIGHT: // RIGHT arrow
                    /* Di chuyển caret sang phải */
                    if (nCurChar < cch) {
                        HideCaret (hWnd);
                        /* Tính toán lại vị trí của caret khi sang phải */
                        ch = pchInputBuf [ nCurChar ];
                        if (ch == 0x0D) {
                            nCaretPosX = 0;
                            nCaretPosY++;
                        }
                        /* Nếu kí tự không phải là enter thì kiểm tra xem phím
                           shift có được nhấn hay không. Nếu có nhấn thì đổi màu
                           và xuất ra màn hình. */
                        else {
                            hdc = GetDC (hWnd);
                            nVirtKey = GetKeyState (VK_SHIFT);
                            if (nVirtKey & SHIFTED) {
                                crPrevText = SetTextColor(hdc,RGB (255, 255, 255));
                                crPrevBk = SetBkColor(hdc,RGB(0,0,0));
                                TextOut(hdc, nCaretPosX, nCaretPosY * dwCharY,
                                     &ch,1);
                                SetTextColor (hdc, crPrevText);//Trả lại màu cũ
                                SetBkColor (hdc, crPrevBk);
                            }
                            GetCharWidth32(hdc,ch,ch,&nCharWidth);
                            ReleaseDC (hWnd, hdc);
                            nCaretPosX = nCaretPosX + nCharWidth;
                        }
                        nCurChar++;
                        ShowCaret (hWnd);
                        break;
                    }
                    break;
                case VK_UP: // Phím mui tên lên
                case VK_DOWN: // Phím mui tên xuống
                    MessageBeep ((UINT) -1);
                    return 0;
                case VK_HOME: // Phím home
                    /* Thiết lập vị trí của caret ở dòng đầu tiên */
                    nCaretPosX = nCaretPosY = 0;
```

```
                    nCurChar = 0;
                    break;
                case VK_END: // Phím end
                    /* Di chuyển về cuối văn bản và lưu vị trí của kí tự xuống
                       dòng cuối */
                    for (i = 0; i < cch; i++) {
                        if (pchInputBuf [i] == 0x0D) {
                            cCR++;
                            nCRIndex = i + 1;//Chỉ ra vtrí kí tự đầu dòng cuối
                        }
                    }
                    nCaretPosY = cCR;
                    /* Chép tất cả các kí tự từ kí tự xuống dòng cuối đến kí tự
                       hiện thời vừa nhập từ bàn phím vào bộ nhớ đệm. */
                    for (i = nCRIndex, j = 0; i < cch; i++, j++)
                    szBuf[j] = pchInputBuf [ i ];
                    szBuf[j] = TEXT('\0');
                    /* Tính vị trí dọc của caret */
                    hdc = GetDC (hWnd);
                    GetTextExtentPoint32(hdc,szBuf, lstrlen(szBuf), &sz);
                    nCaretPosX = sz.cx;
                    ReleaseDC (hWnd, hdc);
                    nCurChar = cch;
                    break;
                default:
                    break;
            }
            SetCaretPos(nCaretPosX, nCaretPosY*dwCharY);
            break;
        case WM_PAINT:
            if (cch == 0)
                break;
            hdc = BeginPaint (hWnd, &ps);
            HideCaret (hWnd);
            SetRect (&rc, 0, 0, dwLineLen, dwClientY);
            DrawText (hdc, pchInputBuf, -1, &rc, DT_LEFT);
            ShowCaret (hWnd);
            EndPaint (hWnd, &ps);
            break;
        case WM_DESTROY:
            PostQuitMessage (0);
            /* Giải phóng buffer */
            GlobalFree ((HGLOBAL) pchInputBuf);
            UnregisterHotKey (hWnd, 0xAAAA);
            break;
        default:
            return DefWindowProc (hWnd, message, wParam, lParam);
    }
    return NULL;
}
```

C. THIẾT BỊ CHUỘT

Trong một môi trường giao tiếp đồ họa, việc sử dụng thiết bị định vị chuột là hết sức cần thiết. Nhờ thiết bị chuột ta có thể di chuyển đến một điểm bất kì trên khung màn hình và thực hiện thao tác kích các nút chuột. Đối với một số ứng dụng đồ họa không quá phức tạp thì công cụ để thực hiện chủ yếu là chuột, riêng đối với các ứng dụng đòi hỏi sự phức tạp khi vẽ thì người ta dùng bút vẽ.

Với Windows thì thiết bị chuột là một thành phần quan trọng, nếu chúng ta bỏ thiết bị chuột thì vẫn khởi động bình thường và các chương trình vẫn có thể chạy được. Tuy nhiên khi đó ta sẽ lúng túng nhiều khi xử lý các ứng dụng trực quan tương tác với người dùng theo tọa độ hay định vị.

Cũng tương tự như bàn phím, thiết bị chuột cũng được dùng để nhập dữ liệu từ người dùng vào ứng dụng nhưng dữ liệu đây không phải là văn bản như khi nhập từ bàn phím mà là các thao tác vẽ hay xử lý các đối tượng đồ họa.

1. Cơ bản về thiết bị chuột trong Windows

Về cơ bản Windows hỗ trợ các loại thiết bị chuột có một nút, hai và ba nút, ngoài ra Windows còn có thể dùng thiết bị khác như joystick hay bút vẽ để bắt chước thiết bị chuột. Các ứng dụng trong Windows thường né việc dùng các nút thứ hai và nút thứ ba, để có thể sử dụng tốt trên các thiết bị chuột chỉ có một nút. Thông thường nút thứ hai của thiết bị chuột phải được dùng cho chức năng gọi thực đơn ngữ cảnh (*context menu*).

Theo nguyên tắc, ta có thể kiểm tra xem có tồn tại thiết bị chuột hay không bằng cách dùng hàm **GetSystemMetrics**.

```
fMouse = GetSystemMetrics(SM_MOUSEPRESENT);
```

Giá trị trả về fMouse là **TRUE** (1) nếu có thiết bị chuột được cài đặt, và ngược lại bằng **FALSE** (0) nếu thiết bị chuột không được cài đặt vào máy.

Ta cũng có thể kiểm tra số nút của thiết bị chuột bằng cách dùng hàm trên theo câu lệnh sau:

```
nButton = GetSystemMetrics(SM_CMOUSEBUTTONS);
```

Giá trị trả về là 0 nếu không cài đặt thiết bị chuột, tuy nhiên dưới Windows 98 giá trị trả về là 2 nếu chưa cài đặt thiết bị chuột.

Khi người dùng di chuyển thiết bị chuột thì Windows cũng di chuyển một ảnh bitmap nhỏ trên màn hình, ảnh này được gọi là con trỏ chuột (*mouse cursor*). Con trỏ chuột này có một điểm gọi là điểm nóng (*hot spot*), điểm nóng này nằm trong hình bitmap, và do người tạo con trỏ chuột quyết định. Khi chúng ta tham chiếu tới một vị trí của con trỏ chuột thì chính là chúng ta đang dùng tọa độ của điểm nóng này.

Windows hỗ trợ một số các con trỏ chuột chuẩn, mà người lập trình chỉ cần gọi ra và dùng chứ không cần phải tạo các tập tin ***.cur**. Ở trạng thái bình thường thì một ứng dụng trong Windows thường dùng con trỏ chuột **IDC_ARROW** được định nghĩa trong **WINUSER.H**, điểm nóng chính là điểm trên cùng của mũi tên. Ngoài ra con trỏ chuột còn cho biết trạng thái của ứng dụng, ví dụ một ứng dụng đang xử lý cần một khoảng thời gian thì con trỏ chuột xuất hiện hình đồng hồ cát (**IDC_WAIT**). Khi đó hầu như các chức năng giao tiếp với người dùng điều phải hoãn đến khi xử lý hoàn thành.

Trong lớp cửa sổ ta có trường định nghĩa con trỏ chuột cho ứng dụng như sau

```
wndclass.hCursor = LoadCursor (NULL, IDC_ARROR);
```

Với thiết bị chuột ta có thể có các hành động như sau:

- *Kích chuột:* nhấn và thả một nút chuột.
- *Kích đúp chuột:* nhấn và thả chuột nhanh (nhấn 2 lần nhanh).
- *Kéo:* di chuyển chuột trong khi vẫn nắm giữ một nút.

2. Xử lý các thông điệp từ thiết bị chuột

Các thông điệp được tạo từ chuột rất khác với thông điệp của bàn phím, một thủ tục cửa sổ sẽ nhận thông điệp chuột bất cứ khi nào thiết bị chuột di chuyển qua cửa sổ hay kích vào trong cửa sổ, thậm chí cả trong trường hợp cửa sổ không được kích hoạt hay không nhận được sự quan tâm. Windows định nghĩa 21 thông điệp được phát sinh từ thiết bị chuột. Trong số đó có đến 11 thông điệp không liên quan đến vùng làm việc (*client area*). Những thông điệp không phải vùng làm việc (*nonclient-area messages*) thường được các ứng dụng bỏ qua không xử lý.

a. Thông điệp chuột trong vùng làm việc

Khi con trỏ chuột di chuyển vào vùng làm việc của một cửa sổ, thủ tục cửa sổ của nó sẽ nhận được thông điệp WM_MOUSEMOVE. Bảng sau mô tả các thao tác làm việc với thiết bị chuột và các thông điệp phát sinh từ nó.

Nút	Nhấn	Thả	Nhấn đúp
Trái	WM_LBUTTONDOWN	WM_LBUTTONUP	WM_LBUTTONDBLCLK
Giữa	WM_MBUTTONDOWN	WM_MBUTTONUP	WM_MBUTTONDBLCLK
Phải	WM_RBUTTONDOWN	WM_MBUTTONUP	WM_RBUTTONDBLCLK

Bảng 4.8 Mô tả thông điệp kích thiết bị chuột

Thủ tục cửa sổ của ứng dụng sẽ nhận được thông điệp của nút chuột giữa nếu máy tính cài thiết bị chuột có 3 nút. Tương tự như vậy với thông điệp thiết bị chuột phải, chúng ta cần có thiết bị chuột dùng 2 nút. Để nhận được thông điệp kích đúp thiết bị chuột thủ tục cửa sổ phải khai báo nhận thông điệp này.

```
Wndclass.style = CS_HREDRAW|CS_VREDRAW|CS_DBLCLKS;
```

Trong thông điệp phát sinh từ thiết bị chuột thì tham số `lParam` sẽ chứa vị trí của thiết bị chuột, 16 byte thấp sẽ chứa giá trị tọa độ x, còn 16 byte cao sẽ chứa giá trị tọa độ của y. Để lấy ra hai giá trị này ta có thể dùng macro là `LOWORD` và `HIWORD`.

Giá trị wParam sẽ cho biết trạng thái của nút nhấn, phím **Shift**, và phím **Ctrl**. Chúng ta có thể kiểm tra các trạng thái này bằng cách dùng bit mặt nạ được định nghĩa trước trong **WINUSER.H**. Các mặt nạ này được bắt đầu bằng tiền tố **MK_xxx** (**Mouse Key**)

MK_LBUTTON	Nút chuột trái nhấn
MK_MBUTTON	Nút chuột giữa nhấn
MK_RBUTTON	Nút chuột phải nhấn
MK_SHIFT	Phím **Shift** được nhấn
MK_CONTROL	Phím **Ctrl** được nhấn

Bảng 4.9 Mô tả trạng thái của chuột và phím nhấn

Ví dụ khi nhận được thông điệp WM_LBUTTONDOWN chúng ta muốn kiểm tra xem phím **Ctrl** có được nhấn hay không bằng cách so giá trị wParam với mặt nạ MK_CONTROL.

```
...
if (wParam & MK_CONTROL) {
   /* Có giữ phím control */
} else {
   /* Không giữ phím control */
}
```

Như chúng ta đã biết khi di chuyển thiết bị chuột qua vùng làm việc thì thông điệp WM_MOUSEMOVE sẽ được gởi đến cho thủ tục cửa sổ đó. Nhưng Windows không phát sinh thông điệp này cho từng pixel trên màn hình mà tuỳ thuộc vào thông số phần cứng của thiết bị chuột được cài đặt và tốc độ làm việc của nó.

Khi chúng ta kích nút chuột trái vào vùng làm việc của một cửa sổ không kích hoạt (*inactive window*) thì Windows sẽ kích hoạt cửa sổ này tức là cửa sổ vừa được kích sau đó truyền thông điệp **WM_LBUTTONDOWN** vào thủ tục **WndProc** của cửa sổ. Khi một cửa sổ nhận được thông điệp **WM_XXXDOWN** thì không nhất thiết phải nhận được thông điệp **WM_XXXUP** hay ngược lại. Điều này được giải thích như sau, khi người dùng kích trái vào một cửa sổ và giữ luôn nút chuột vừa kích rồi kéo thiết bị chuột đến một vùng thuộc phạm vi của cửa sổ khác mới thả. Khi đó cửa sổ đầu tiên sẽ nhận được thông điệp nhấn chuột **WM_LBUTTONDOWN** và cửa sổ thứ hai sẽ nhận được thông điệp nhả thiết bị chuột **WM_LBUTTONUP**. Tuy nhiên tình huống trên sẽ không xuất hiện với hai trường hợp ngoại lệ sau:

- Thủ tục **WndProc** của một cửa sổ đang thực hiện việc bắt giữ thiết bị chuột (*mouse capture*), đối với tình trạng này thì cửa sổ tiếp tục nhận được thông điệp chuột cho dù con trỏ chuột có di chuyển ra ngoài vùng làm việc của cửa sổ. Kiểu này thường xuất hiện trong các ứng dụng vẽ hay thao tác đối tượng đồ họa, ví dụ khi ta vẽ một đường thẳng dài và kéo ra ngoài vùng làm việc của cửa sổ, thì khi đó cửa sổ vẽ sẽ bắt giữ toạ độ của thiết bị chuột để tạo đường thẳng và có thể cho thanh cuộn cuộn theo.
- Nếu xuất hiện hộp thông tin trạng thái (*modal*) của hệ thống, thì không có chương trình nào khác nhận được thông điệp của thiết bị chuột. Hộp thoại trạng thái hệ thống và hộp thoại trạng thái của ứng dụng ngăn cản việc chuyển qua cửa sổ khác trong một ứng khi nó chưa giải quyết xong hay vẫn còn trạng thái kích hoạt (*active*).

b. Thông điệp của thiết bị chuột ngoài vùng làm việc

Với các thông điệp của thiết bị chuột vừa tìm hiểu trong phần trước đều được phát sinh khi thiết bị chuột nằm trong vùng làm việc của cửa sổ. Khi di chuyển con trỏ chuột ra khỏi vùng làm việc của cửa sổ nhưng vẫn ở trong phạm vi của cửa sổ thì khi đó các thông điệp của thiết bị chuột sẽ được phát sinh dạng thông điệp của thiết bị chuột ngoài vùng làm việc (*nonclient-area*). Ngoài vùng làm việc của một cửa sổ là cửa sổ thanh tiêu đề, thực đơn, và thanh cuộn của cửa sổ.

Nói chung với các thông điệp của thiết bị chuột phát sinh từ ngoài vùng làm việc thì chúng ta không quan tâm lắm, thay vào đó ta giao phó cho hàm mặc định xử lý là **DefWindowProc** thực hiện. Điều này cũng giống như là thông điệp bàn phím hệ thống mà ta đã tìm hiểu trong các phần trước.

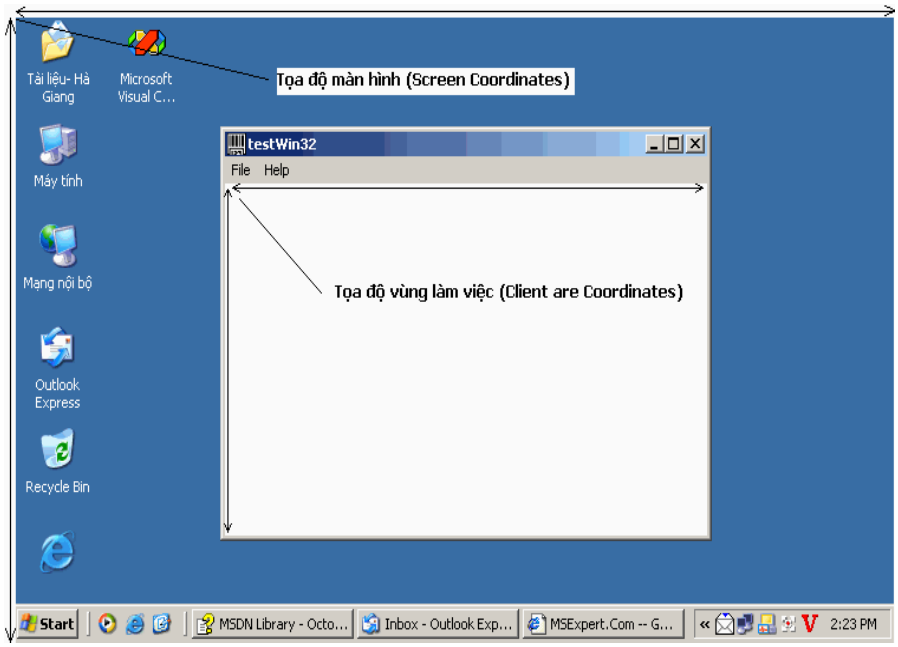
Cũng tương tự như thông điệp xuất phát từ vùng làm việc, các thông điệp ngoài vùng làm việc được định nghĩa với từ **NC(nocilent area)** vào sau dấu "_", ta có bảng mô tả các thông điệp phát sinh từ ngoài vùng làm việc như sau.

Nút	Nhấn	Thả	Nhấn đúp
Trái	WM_NCLBUTTONDOWN	WM_NCLBUTTONUP	WM_NCLBUTTONDBLCLK
Giữa	WM_NCMBUTTONDOWN	WM_NCMBUTTONUP	WM_NCMBUTTONDBLCLK
Phải	WM_RBUTTONDOWN	WM_NCRBUTTONDOWN	WM_NCRBUTTONDBLCLK

Bảng 4.10 Mô tả các thông điệp chuột ngoài vùng làm việc

Các tham số lParam và wParam cũng hơi khác so với các thông điệp thiết bị chuột phát sinh trong vùng làm việc. Với tham số lParam của thông điệp phát sinh từ ngoài vùng làm việc sẽ chỉ ra vị trí ngoài

vùng làm việc nơi mà thiết bị chuột di chuyển hay kéo tới. Vị trí này được định danh bởi các giá trị định nghĩa trong **WINUSER.H** được bắt đầu với **HT** (viết tắt cho *hit-test*).



Hình 4.2 Minh họa tọa độ màn hình và tọa độ vùng làm việc

Tham số `lParam` sẽ chứa tọa độ x ở 16 byte thấp và tọa độ y ở 16 byte cao. Tuy nhiên, đây là tọa độ màn hình, không phải là tọa độ vùng làm việc giống như thông điệp phát sinh từ vùng làm việc. Do đó chúng ta phải chuyển về tọa độ vùng làm việc để xử lý tiếp nếu cần.

Để chuyển từ tọa độ màn hình sang tọa độ làm việc hay ngược lại từ tọa độ làm việc sang tọa độ màn hình ta dùng hai hàm tương ứng được Windows cung cấp như sau:

```
ScreenToClient(hwnd, &pt);
ClientToScreen(hwnd, &pt);
```

`pt` là biết cấu trúc **POINT**, hai hàm trên sẽ nhận tham chiếu đến biến `pt` do đó sau khi gọi hàm ta sẽ được giá trị pt tương ứng ở tọa độ mới.

3. Ví dụ minh hoạ thiết bị chuột

Đoạn chương trình thực hiện chức năng vẽ tự do bằng các thao tác: Nhấn chuột trái để vẽ một đường thẳng từ vị trí con trỏ của bút vẽ đến vị trí chuột trái kích, ngoài ra ta cũng có thể vẽ bằng cách nhấn phím trái và giữ luôn sau đó ta kéo chuột đến vị trí bất kì rồi thả. Nếu ta nhấn chuột phải thì sẽ đổi màu của bút vẽ.

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam) {
    HDC hdc;
    static POINT oldPoint;
    static int iC;
    int WIDTH_PEN = 2;
    HPEN oPen,pen;
    COLORREF Col[] = {RGB(0, 0, 0), RGB(255,0,0), RGB(0, 255, 0),
                      RGB(0, 0, 255), RGB(255, 255, 0)};
    POINT point;
    TCHAR str [255];
    switch (message) {
        case WM_LBUTTONDOWN:
            /* Vẽ đường thẳng từ vị trí trước đó đến vị trí chuột hiện tại*/
            hdc = GetDC (hWnd);
            pen = CreatePen (PS_SOLID,WIDTH_PEN,Col [ iC]);
            oPen = (HPEN) SelectObject (hdc,pen);//hàm trả về bút vẽ trước đó
            point.x = LOWORD (lParam);
            point.y = HIWORD (lParam);
            MoveToEx (hdc, oldPoint.x, oldPoint.y, NULL);
            LineTo (hdc, point.x, point.y);
```

```
            oldPoint = point;
            /* Chọn lại bút vẽ trước đó và hủy bút vừa tạo*/
            SelectObject (hdc, oPen);
            DeleteObject (pen);
            ReleaseDC (hWnd, hdc);
            break;
        case WM_RBUTTONDOWN:
            /* Chuyển index của bảng màu sang vị trí tiếp theo,
               nếu cuối bảng màu thì quay lại màu đầu tiên*/
            iC = (iC+1) % (sizeof (Col) / sizeof (COLORREF));
            break;
        case WM_MOUSEMOVE:
            /* Xuất toạ độ chuột hiện thời lên thanh tiêu đề*/
            sprintf(str,"Toa do chuot x = %d, To do y = %d",
                    LOWORD(lParam), HIWORD(lParam));
            SetWindowText (hWnd, str);
            /* Kiểm tra xem có giữ phím chuột trái hay không*/
            if (wParam & MK_LBUTTON) {
                hdc = GetDC (hWnd);
                pen = CreatePen (PS_SOLID,WIDTH_PEN,Col [ iC ]);
                oPen = (HPEN) SelectObject (hdc, pen);
                point.x = LOWORD (lParam);
                point.y = HIWORD (lParam);
                MoveToEx (hdc, oldPoint.x, oldPoint.y, NULL);
                LineTo (hdc, point.x, point.y);
                oldPoint = point;
                SelectObject (hdc, oPen);
                DeleteObject (pen);
                ReleaseDC (hWnd, hdc);
            }
            break;
        case WM_DESTROY:
            PostQuitMessage (0);
            break;
        default:
            return DefWindowProc (hWnd, message, wParam, lParam);
    }
    return 0;
}
```

D. BỘ ĐỊNH THỜI GIAN

Như chúng ta đã biết Windows cung cấp cho ta các thông tin thông qua dạng thông điệp như là thông điệp bàn phím, thông điệp phát sinh từ thiết bị chuột. Và ngoài ra thì một thông điệp cũng rất hữu dụng là thông điệp thời gian. Khi viết một chương trình chúng ta có thể yêu cầu hệ điều hành gởi một thông điệp cảnh báo theo từng khoảng thời gian nhất định để chúng ta có thể làm một số xử lý cần thiết. Thông điệp này được gởi từ hệ điều hành đến chương trình thông qua một bộ định thời gian (*Timer*) và thông điệp được phát sinh là `WM_TIMER`. Việc dùng chức năng này rất đơn giản, ta khai báo một bộ định thời gian và thiết lập thông số khoảng thời gian để Windows phát sinh thông điệp **Timer** cho ứng dụng. Khi đó ứng dụng chỉ cần xử lý thông điệp `WM_TIMER` trong hàm xử lý cửa sổ **`WinProc`**.

1. Bộ định thời gian và vấn đề đồng bộ

Theo lý thuyết thông điệp thời gian do Windows cung cấp là chính xác đến mili giây nhưng thực tế không hoàn toàn như vậy. Sự chính xác còn phụ thuộc vào đồng hồ của hệ thống và các hoạt động hiện thời của chương trình. Nguyên nhân của vấn đề là thông điệp thời gian `WM_TIMER` có độ ưu tiên thấp, như thông điệp tô vẽ lại màn hình `WM_PAINT`, thông thường chúng phải chờ cho xử lý xong các thông điệp khác.

2. Khởi tạo bộ định thời gian

Không như thông điệp xuất phát từ bàn phím và chuột, được Windows gởi tự động vào hàng đợi thông điệp của ứng dụng. Đối với thông điệp thời gian phải được khai báo trước bằng hàm **SetTimer**, sau khi khai báo hàm này thì Windows sẽ gởi thông điệp WM_TIMER điều đặn vào hàng đợi của ứng dụng.

Hàm **SetTimer** được khai báo như sau:

```
UINT_PTR SetTimer(HWND hWnd, UINT_PTR nIDEvent,
                  UINT uElapse, TIMERPROC lpTimerFunc);
```

Trong đó ý nghĩa các tham số được mô tả:

- hWnd: Định danh của cửa sổ khai báo dùng bộ định thời gian.
- nIDEvent: Định danh của bộ định thời gian.
- nElapse: Là khoảng thời gian nghỉ giữa hai lần gởi thông điệp
- lpTimerFunc: Hàm sẽ xử lý khi thông điệp WM_TIMER phát sinh, nếu chúng ta khai báo là NULL thì Windows sẽ gởi thông điệp WM_TIMER vào hàng đợi thông điệp của cửa sổ tương ứng.

Khi không còn dùng bộ định thời gian nữa hay kết thúc ứng dụng ta gọi hàm **KillTimer**, hàm này được khai báo:

```
BOOL KillTimer(HWND hWnd, UINT_PTR uIDEvent);
```

- hWnd: Định danh của cửa sổ dùng bộ định thời gian
- uIDEvent: Định danh của bộ định thời gian.

3. Ví dụ về bộ định thời gian

a. Dùng thông điệp WM_TIMER

Đoạn chương trình minh họa việc sử dụng bộ định thời gian trong chương trình. Cứ mỗi 0.5 giây thì chương trình phát sinh tự động ngẫu nhiên một vòng tròn trên màn hình.

```
#include <time.h>
#include "stdio.h"
#define MAX_POINT 10000
#define IDT_TIMER1 1
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam) {
    PAINTSTRUCT ps;
    HDC hdc;
    static int NumCir = 0;
    static POINT point [ MAX_POINT ];
    int r = 5, i;
    HPEN pen, oldPen;
    RECT rc;
    TCHAR str [255];
    /* Xử lý thông điệp*/
    switch (message) {
        case WM_CREATE:
            /* Khai báo dùng bộ định thời gian trong ứng dụng*/
            SetTimer(hWnd, IDT_TIMER1, 500, (TIMERPROC)NULL);
            /* Khởi động hàm ngẫu nhiên*/
            srand ((unsigned) time(NULL));
            break;
        case WM_PAINT:
            hdc = BeginPaint (hWnd, &ps);
            pen = CreatePen (PS_SOLID, 2, RGB (255,0,0));
            oldPen = (HPEN) SelectObject (hdc, pen);
            /* Vẽ các vòng tròn với tâm lưu trong point và bán kính r */
            for(i = 0; i < NumCir; i++)
            Arc(hdc, point[i].x-r, point[i].y-r, point[i].x+r,
                point[i].y+r, point[i].x+r, point[i].y, point[i].x+r, point[i].y);
            /* Lấy lại bút vẽ trước đó và hũy bút vẽ vừa tạo*/
            SelectObject (hdc, oldPen);
            DeleteObject (pen);
            EndPaint (hWnd, &ps);
            break;
```

```
        case WM_TIMER:
            /* Lấy thông tin của vùng làm việc*/
            GetClientRect (hWnd, &rc);
            /* Phát sinh ngẫu nhiên một vòng tròn */
            point[NumCir].x = rand() % (rc.right - rc.left);
            point[NumCir].y = rand() % (rc.bottom - rc.top);
            NumCir++;
            /* Hiển thị số vòng tròn đã sinh ra trên thanh tiêu đề */
            sprintf (str,"So vong tron: %d", NumCir);
            SetWindowText (hWnd, str);
            /* Làm bất hợp lệ vùng làm việc & yêu cầu vẽ lại */
            InvalidateRect (hWnd, &rc, FALSE);
            break;
        case WM_DESTROY:
            /* Hủy bỏ sử dụng bộ định thời gian*/
            KillTimer (hWnd, IDT_TIMER1);
            PostQuitMessage (0);
            break;
        default:
            return DefWindowProc (hWnd, message, wParam, lParam);
    }
    return 0;
}
```

b. Dùng hàm xử lý

Đoạn chương trình sau cũng khai báo sử dụng một bộ định thời gian, nhưng khai báo trực tiếp, tức là khi hết thời gian chờ thay vì truyền thông điệp **WM_TIMER** thì Windows gọi hàm **TimerProc** thực hiện.

Chương trình khi thực thi sẽ xuất ra một dạng đồng hồ điện tử theo dạng "giờ: phút:giây".

```
#include <time.h>
#include "stdio.h"
#define IDT_TIMER1 1
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam) {
    PAINTSTRUCT ps;
    HDC hdc;
    /* Khai báo biến lưu các giá trị không gian*/
    struct tm *newtime;
    time_t CurTime;
    TCHAR str [255];
    RECT rc;
    /* Biến LOGFONT để tạo font */
    LOGFONT lf;
    HFONT oldFont, font;
    COLORREF color = RGB (255, 0, 0), oldColor;
    switch (message) {
        case WM_CREATE:
        /* khởi tạo bộ định thời gian, và khai báo hàm xử lý Timer */
        SetTimer (hWnd, IDT_TIMER1, 1000, (TIMERPROC) TimerProc);
        break;
        case WM_PAINT:
             hdc = BeginPaint (hWnd, &ps);
             /* Lấy giờ đồng hồ hệ thống */
             time(&CurTime);
             newtime = localtime (&CurTime);
             GetClientRect (hWnd, &rc);
             /* Tạo chuỗi xuất ra màn hình */
             sprintf(str,"Gio hien tai: %d gio: %d phut: %d giay",
                 newtime->tm_hour, newtime->tm_min, newtime->tm_sec);
             /* Thiết lập màu kí tự xuất */
             oldColor = SetTextColor (hdc, color);
             /* Tạo font riêng để dùng */
```

```
            memset (&lf, 0, sizeof (LOGFONT));
            lf.lfHeight = 50;
            strcpy (lf.lfFaceName, "Tahoma");
            font = CreateFontIndirect (&lf);
            oldFont = (HFONT) SelectObject (hdc,font);
            /* Xuất ra màn hình */
            DrawText (hdc, str, strlen(str), &rc,
               DT_CENTER | DT_VCENTER | DT_SINGLELINE);
            /* Lấy lại các giá trị mặc định */
            SetTextColor (hdc,oldColor);
            SelectObject (hdc,oldFont);
            DeleteObject (font);
            EndPaint (hWnd, &ps);
            break;
        case WM_DESTROY:
            PostQuitMessage (0);
            break;
        default:
            return DefWindowProc (hWnd, message, wParam, lParam);
    }
    return 0;
}

/* Hàm xử lý của Timer */
VOID CALLBACK TimerProc(HWND hwnd, UINT uMsg, UINT_PTR idEvent, DWORD dwTime)
{
    /* Hàm này đơn giản yêu cầu tô lại vùng làm việc */
    RECT rc;
    GetClientRect (hwnd, &rc);
    InvalidateRect (hwnd, &rc, TRUE);
}
```

CHƯƠNG V: XỬ LÝ VĂN BẢN VÀ FONT CHỮ

A. MỞ ĐẦU

Một trong các đặc tính nổi bật nhất của Windows là giao diện giao tiếp với người dùng. Nhờ đặc tính này, nhiều dạng dữ liệu thông tin khác nhau được máy tính hỗ trợ khi xuất ra màn hình, máy in, … Trong đó, xuất văn bản ra vùng làm việc của các ứng dụng Win32 là hình thức phổ biến nhất trong giao diện đồ họa.

Ở các chương trước, các thao tác xuất thông tin chỉ được thực hiện thông qua các cửa sổ thông báo, hộp thoại và các phần tử điều khiển. Chương này sẽ trình bày cách thể hiện nội dung văn bản trên vùng làm việc của cửa sổ thông qua các hàm Win32® API.

Phần cuối chương sẽ trình bày cách khởi tạo, chọn và xử lý các dạng font chữ khác nhau. Nhờ khả năng này, việc thể hiện các đoạn văn bản sẽ trở nên sinh động và trực quan hơn.

B. XỬ LÝ VĂN BẢN

Xử lý văn bản là công việc phổ biến nhất trong các thao tác đồ họa. Chúng được sử dụng theo các định dạng và cách thức khác nhau trong các ứng dụng xử lý tài liệu, bảng biểu, cơ sở dữ liệu và hỗ trợ thiết kế bằng máy tính (**CAD** - *Computer Aided Design*).

Tập hợp các hàm Win32® API xử lý văn bản được phân thành hai nhóm chính: ***Nhóm các hàm định dạng*** – chuẩn bị cho thao tác xuất dữ liệu, và ***nhóm các hàm thực hiện thao tác hiển thị***. Chúng ta sẽ bắt đầu với việc tìm hiểu các hàm hiển thị.

1. Hiển thị văn bản

Để hiện thị nội dung văn bản trên các thiết bị xuất, dựa vào từng trường hợp thể hiện khác nhau, ta dùng các hàm Win32 API khác nhau. Các hàm này phụ thuộc vào font chữ, thuộc tính của thiết bị ngữ cảnh **DC** (*Device Context)* và khoảng cách ký tự thể hiện.

Hàm phổ biến nhất thực hiện thao tác xuất một chuỗi ký tự văn bản, sử dụng font chữ, màu chữ và màu nền hiện hành là:

```
BOOL TextOut(HDC hDC, int nXStart, int nYStart,
             LPCTSTR lpString, int cbString);
```

Hàm này thực hiện thao tác xuất chuỗi ký tự xác định bởi con trỏ `lpString` ra DC, với chiều dài được xác định bởi `cbString` (không phụ thuộc vào ký tự `NULL` đánh dấu kết thúc chuỗi).

Hai trường `nXStart` và `nYStart` là vị trí gốc của chuỗi hiển thị, xác định theo tọa độ logic của vùng làm việc cửa sổ, và thường là điểm gốc trên bên trái của vùng hiển thị chuỗi. Chúng ta sẽ bàn kỹ hơn khi tìm hiểu về canh lề văn bản trong phần 2.

Nếu thao tác xuất chuỗi thực hiện thành công, hàm trả về giá trị khác 0. Ngược lại, giá trị trả về bằng 0.

Khi cần trình bày văn bản theo tuần tự từng cột, ta dùng hàm **TabbedTextOut** sau:

```
LONG TabbedTextOut(HDC hDC, int nX, int nY,
                   LPCTSTR lpString, int nCount, int nNumTabs,
                   LPINT lpnTabStopPositions, int nTabOrigin);
```

Nếu trong chuỗi ký tự có các ký tự tab (**'\t'** hoặc `0x09`), hàm **TabbedTextOut** sẽ chuyển các ký tự tab vào dãy các vị trí "dừng" tương ứng. Số lượng các tab dừng được xác định bởi `nNumTabs`, và `lpnTabStopPositions` là dãy vị trí các tab dừng theo đơn vị tính pixels. Ví dụ, nếu độ rộng trung bình của mỗi ký tự là 8 pixels, và mỗi tab dừng cần đặt cách nhau 5 ký tự, dãy các tab dừng sẽ phải lần lượt có giá trị 40, 80, 120, …. Tuy nhiên, các giá trị này không nhất thiết phải là bội số của nhau.

Nếu biến `nNumTabs` hoặc `lpnTabStopPositions` có giá trị là 0 và NULL, các tab dừng được đặt cách nhau từng 8 ký tự. Nếu `nNumTabs` bằng 1, `lpnTabStopPositions` trỏ đến giá trị xác định một dãy tăng tuần hoàn là bội số của dãy này. Ví dụ, nếu `nNumTabs` bằng 1, và `lpnTabStopPositions` bằng 30, ta sẽ có dãy tab dừng tại vị trí 30, 60, 90, … pixels.

Trường `nTabOrigin` xác định tọa độ theo trục **x** của điểm bắt đầu tính khoảng cách tới các tab. Giá trị này không nhất thiết phải là vị trí đầu tiên của chuỗi, có thể chọn trùng hoặc không.

Hàm trả về kích thước chuỗi hiển thị, theo đơn vị logic, nếu thành công. Ngược lại, hàm trả về 0. Trong đó, chiều cao chuỗi là **WORD** cao của biến kiểu **LONG**, chiều rộng là **WORD** thấp.

Một hàm xuất văn bản khác tương tự hàm **TextOut** là hàm **ExtTextOut:**

```
BOOL ExtTextOut(HDC hDC, int X, int Y,
```

```
                    UINT fuOptions, CONST RECT *lprc, LPCTSTR lpString,
                    UINT cbCount, CONST INT *lpDx);
```

Biến lprc là một con trỏ đến cấu trúc **RECT**, trong đó nội dung văn bản hiển thị sẽ bị cắt vào vùng hình chữ nhật tương ứng nếu fuOptions được thiết lập là ETO_CLIPPED, hoặc là vùng nền hình chữ nhật sẽ được tô bởi màu nền nếu fuOptions là ETO_OPAQUE.

Trường lpDx là một dãy số nguyên xác định khoảng cách giữa các ký tự liên tiếp trong chuỗi. Nó cho phép một chương trình tạo khoảng cách rộng hoặc hẹp giữa các ký tự, điều mà đôi lúc cần thiết trong việc điều chỉnh các từ trong văn bản theo độ rộng cột. Giá trị lpDx có thể là NULL tương ứng với chế độ mặc định cho khoảng cách này.

Tương tự hàm **TextOut**, hàm **ExtTextOut** trả về giá trị khác 0 nếu thành công. Ngược lại, giá trị trả về bằng 0.

Một hàm mức ở mức cao hơn để xuất văn bản là hàm **DrawText:**

```
int DrawText(HDC hDC, LPCTSTR lpString, int nCount,
             LPRECT lpRect, UINT uFormat);
```

Cũng như các hàm xuất văn bản khác, hàm **DrawText** xuất chuỗi xác định bởi con trỏ lpString có độ dài nCount. Tuy nhiên, với chuỗi có ký tự kết thúc là NULL, nếu nCount bằng -1, hàm sẽ tự động tính toán chiều dài của chuỗi.

Biến lpRect trỏ đến cấu trúc **RECT** của hình chữ nhật (theo toạ độ logic) mà trong đó văn bản thể hiện theo định dạng được thiết lập trong uFormat.

Nếu uFormat bằng 0, nội dung văn bản sẽ được hiển thị theo từng dòng từ trên xuống dưới. Mỗi dòng mới được xác định thông qua ký tự về đầu dòng **CR** (*carriage return*, bằng '\r' hoặc 0x0D) hoặc ký tự xuống dòng **LF** (*linefeed*, bằng '\n' hoặc 0x0A) có trong văn bản. Phần văn bản bên ngoài hình chữ nhật lpRect sẽ bị cắt bỏ.

Giá trị uFormat bằng 0 cũng chính là giá trị cờ canh lề trái (DT_LEFT). Ngoài ra, ta có thể thiết lập các cờ canh lề phải (DT_RIGHT), và canh lề giữa (DT_CENTER) cho văn bản.

Để loại bỏ chức năng điều khiển của các ký tự **CR** và **LF**, cần thêm vào cờ DT_SINGLELINE. Nếu thiết lập DT_SINGLELINE, ta cũng có thể chỉ định vị trí của dòng hiển thị ở phía trên (DT_TOP), phía dưới (DT_BOTTOM), hoặc ở chính giữa (DT_VCENTER) trong vùng hình chữ nhật.

Trong trường hợp hiển thị nhiều dòng văn bản, Windows chỉ ngắt dòng khi gặp ký tự CR và LF. Để ngắt dòng dài hơn kích thước hình chữ nhật hiển thị, cần thiết lập cờ DT_WORDBREAK. Nếu không muốn Windows cắt bỏ các phần dư ra khi vẽ chữ vượt quá phạm vi khung chữ nhật, ta thêm cờ DT_NOCLIP. Nếu muốn ký tự **tab** ('\t' hoặc 0x09) được diễn dịch thành ký tự phân cột, cần thêm cờ DT_EXPANDTABS. Giá trị mặc định của tab là 8 khoảng trắng. Cờ DT_TABSTOP được dùng để đặt lại giá trị tab. Trong trường hợp này, byte cao của word thấp (bits 15-8) của uFormat sẽ chứa giá trị tab cần thay thế.

2. Định dạng văn bản

Dựa vào đặc trưng các thành phần hiển thị, các hàm định dạng văn bản phân làm ba nhóm liên quan đến ***thuộc tính của DC***, ***độ rộng ký tự*** và ***kích thước chuỗi ký tự hiển thị***.

Việc thiết lập thuộc tính định dạng văn bản cho DC được thực hiện thông qua các hàm canh lề văn bản, thiết lập khoảng cách ký tự, xác định màu nền và màu văn bản. Cùng với các hàm này, Windows cũng cung cấp các hàm cho biết thuộc tính hiện hành tương ứng cho DC.

Trong các hàm về thuộc tính DC, biến đầu tiên luôn là **handle** của DC hiện hành. Xét hàm thiết lập màu chữ và màu nền:

```
COLORREF SetTextColor(HDC hDC, COLOREF crColor);
COLORREF SetBkColor(HDC hDC, COLORREF crColor);
```

Biến crColor xác định màu cần thiết lập. Nếu thành công, hàm trả về màu chữ (màu nền) trước khi được thiết lập. Nếu không, hàm trả về giá trị cờ CLR_INVALID. Ngoài ra, để xác định màu chữ và màu nền hiện hành, ta dùng hai hàm sau:

```
COLORREF GetTextColor(HDC hDC);
COLORREF GetBkColor(HDC hDC);
```

Nếu hàm thực hiện thành công, ta xác định được màu hiện hành. Nếu không, giá trị trả về là CLR_INVALID.

Khi vẽ chữ, Windows sử dụng hai chế độ: chế độ trong suốt (**TRANSPARENT**) và chế độ mờ (**OPAQUE**). Ở chế độ trong suốt, màu nền sẽ không được sử dụng đến, chữ vẽ ra đè lên nền hiện hành. Ở chế độ mờ, trước khi vẽ chữ, nền sẽ được xoá đi với màu nền được thiết lập bởi hàm **SetBkColor**:

```
int SetBkColor(HDC hDC, int iBkMode);
```

Với iBkMode là chế độ nền TRANSPARENT hoặc OPAQUE (chế độ mặc định của Windows là OPAQUE). Nếu thành công, hàm trả về chế độ nền trước khi được thiết lập. Ngược lại, giá trị trả về là 0. Để biết chế độ nền hiện tại, ta dùng hàm:

```
int GetBkMode(HDC hDC);
```

Hàm trả về giá trị TRANSPARENT hoặc OPAQUE, nếu thành công. Ngược lại, giá trị trả về là 0.

Để xác lập vị trí chuỗi văn bản hiển thị dựa trên điểm gốc nXStart, nYStart (xem phần 5.2.1) ta dùng hàm **SetTextAlign:**

```
UINT SetTextAlign(HDC hDC, UINT fMode);
```

Khi đó, điểm gốc nXStart ở cạnh bên trái khung chữ nhật nếu fMode là TA_LEFT. Ký tự đầu chuỗi sẽ hiển thị từ điểm gốc này. Đây cũng là giá trị mặc định của Windows. Nếu fMode bằng TA_RIGHT, vị trí chuỗi được tính từ bên phải, tức ký tự cuối chuỗi hiển thị tại điểm gốc, và ngược lại cho đến ký tự đầu tiên. Nếu fMode bằng TA_CENTER, vị trí giữa chuỗi chính là điểm gốc.

Tương tự, để thiết lập vị trí hiển thị chuỗi theo phương đứng, các cờ TA_TOP, TA_BOTTOM, và TA_BASELINE được dùng tương ứng điểm gốc nYStart ở trên, giữa và dưới dòng văn bản hiển thị. Đối với Windows thì giá trị mặc định theo phương đứng là TA_TOP.

Nếu gọi hàm **SetTextAlign** với cờ TA_UPDATE, Windows sẽ không sử dụng điểm gốc nXStart, nYStart trong hàm xuất văn bản **TextOut**, thay vào đó là vị trí được thiết lập trước đó bởi hàm **MoveToEx** hoặc **LineTo**, hoặc một hàm thay đổi vị trí khác. Cờ TA_UPDATE cũng cập nhật điểm gốc về đầu chuỗi (nếu dùng TA_LEFT) và về cuối chuỗi (nếu dùng TA_RIGHT) cho lần gọi kế tiếp. Điều này cần thiết cho việc hiển thị nhiều dòng văn bản với hàm **TextOut**. Nếu cờ TA_CENTER được thiết lập, vị trí của nXStart vẫn như cũ sau khi hàm **TextOut** được gọi.

Để biết chế độ canh lề văn bản hiện tại, ta dùng hàm:

```
UINT GetTextAlign(HDC hDC);
```

Nếu thành công, hàm trả về cờ tương ứng của canh lề văn bản hiện hành. Ngược lại, giá trị trả về là GDI_ERROR.

Ví dụ sau đây trình bày cách thức xác định các dạng canh lề theo phương ngang:

```
switch ((TA_LEFT | TA_RIGHT | TA_CENTER) & GetTextAlign(hDC)) {
case TA_LEFT:
.
.
.
case TA_RIGHT:
.
.
.
case TA_CENTER:
.
.
.
}
```

Ví dụ tiếp theo sử dụng hàm **SetTextAlign** để cập nhật vị trí hiển thị hiện thời khi hàm **TextOut** được gọi. Trong ví dụ này, biến cArial là một số nguyên cho biết số **font Arial**.

```
UINT uAlignPrev;
char szCount[8];
uAlignPrev = SetTextAlign(hdc, TA_UPDATECP);
MoveToEx(hdc, 10, 50, (LPPOINT) NULL);
TextOut(hdc, 0, 0, "Number of Arial fonts: ", 23);
itoa(cArial, szCount, 10);
TextOut(hdc, 0, 0, (LPSTR) szCount, strlen(szCount));
SetTextAlign(hdc, uAlignPrev);
```

Một thuộc tính khác của DC ảnh hưởng đến cách vẽ chuỗi là khoảng cách giữa các ký tự trong chuỗi hiển thị. Khoảng cách mặc định của Windows là 0, khi đó các ký tự được hiển thị liên tiếp nhau. Để thay đổi khoảng cách giữa các ký tự, ta dùng hàm:

```
int SetTextCharacterExtra(HDC hDC, int nCharExtra);
```

Trong đó, nCharExtra là khoảng cách theo đơn vị logic thiết lập giữa các ký tự. Nếu thành công, hàm trả về khoảng cách trước khi được thiết lập. Ngược lại, giá trị trả về là 0x80000000. Để biết khoảng cách hiện tại, ta dùng hàm:

```
int GetTextCharacterExtra(HDC hDC);
```

Nếu thành công, giá trị trả về cho biết khoảng cách hiện tại. Ngược lại, giá trị trả về là 0x80000000.

Ngoài ra, Windows còn hỗ trợ các hàm cho biết độ rộng ký tự và kích thước chuỗi hiển thị. Đây là các hàm cấp cao, sử dụng trong việc trình bày văn bản với các kiểu font khá phức tạp. Trong chương này, chúng ta chỉ đề cập đến một số hàm như **GetTextMetrics** (phần C.2) và **GetTextExtentPoint32** (phần C.5).

C. FONT CHỮ

Trong Windows, khi trình bày văn bản, các ký tự được thể hiện theo nhiều dạng khác nhau. Đây là một trong những đặc trưng cơ bản của giao diện đồ họa - người dùng (**GUI** – *Graphical User Interface*). Để thực hiện điều này, Windows hỗ trợ nhiều dạng font chữ khác nhau. Trong phần này, chúng ta tìm hiểu các vấn đề chính về các font chữ, cũng như cách sử dụng chúng để trình bày văn bản.

1. Khái niệm font trong Windows

Một font chữ là một tập hợp các ký tự và ký hiệu cùng dạng, thể hiện qua kiểu chữ, loại chữ và kích cỡ chữ.

Kiểu chữ xác định các đặc trưng về ký tự và ký hiệu trong font chữ, ví dụ độ rộng, nét chữ dày hoặc mảnh, có chân (có gạch ngang hoặc các nét cong mảnh ở đầu các ký tự) hay không.

Loại chữ xác định độ đậm nhạt (trọng lượng) và độ nghiêng của dạng font thể hiện. Chia làm ba loại sau: roman, opaque, và italic. Font roman là dạng font chữ có trọng lượng trung bình, thường dùng trong in ấn. Font opaque là dạng được biến đổi ngiêng của font roman. Font italic là dạng font được thiết kế theo dạng nghiêng chuyên biệt.

Kích thước font chữ được tính như là khoảng cách từ chặn dưới của một ký tự có chân đến chặn trên của một ký tự hoa, và được tính theo đơn vị điểm (khoảng 0,013817 của 1 inch).

Một tập các font với một số kích cỡ và trọng lượng khác nhau nhưng cùng một kiểu loại được xem là một họ font chữ.

Các font chữ trong Windows được chia thành hai nhóm, gọi là font GDI và font thiết bị. Font GDI được lưu trữ dưới dạng tập tin trên đĩa, trong khi font thiết bị được thiết kế sẵn trong thiết bị xuất tương ứng, ví dụ máy in. Không như font GDI, khi in bằng font thiết bị, Windows không cần định dạng font, mà chỉ cần gởi nội dung dữ liệu trực tiếp đến thiết bị. Người dùng chọn trực tiếp font từ máy in để in. Đối với font GDI, Windows chuyển văn bản thành ảnh bitmap sau đó chuyển đến máy in để in. Như vậy, lợi điểm của font GDI là độc lập với thiết bị xuất, tuy nhiên tốc độ xử lý in chậm hơn.

Các font GDI thông thường gồm có: **font bitmap**, **font vector** và **font TrueType**. Font bitmap được cấu thành từ ma trận pixel, có lợi điểm là hiển thị nhanh, nhưng hạn chế trong việc thể hiện với các kích thước khác nhau (do chỉ phóng to – thu nhỏ, và khi phóng trông rất thô). Font vector thể hiện các ký tự như là tập hợp các nét vẽ (sử dụng các hàm GDI), do đó linh động và co giản hơn font bitmap, tuy nhiên được vẽ khá chậm và đường nét cũng không được mềm mại cho lắm.

Font TrueType là dạng font được sử dụng phổ biến nhất hiện nay, được lưu trữ dưới dạng một tập các điểm ảnh, kết hợp một số thuật toán biến đổi. Do đó thể hiện sắc xảo các đường nét trên màn hình và máy in. Chúng ta sẽ khảo sát các hàm xử lý cho các font chữ dạng này trong phần C.3. Còn bây giờ chúng ta tìm hiểu cách sử dụng các font có sẵn của hệ thống.

2. Sử dụng font định nghĩa sẵn

Khi ta gọi các hàm vẽ chữ **TextOut**, **TabbedTextOut**, **ExtTextOut** hoặc **DrawText**, Windows sẽ sử dụng font chữ đang được chọn trong DC để hiện thị nội dung văn bản. Ta có thể dùng một số font được Windows định nghĩa sẵn, thường gọi là font hệ thống (phân biệt các font do người dùng tạo trong quá trình xử lý văn bản – xem phần 5.3.3). Tên macro của các font này thể hiện trong bảng sau:

MACRO	FONT
ANSI_FIXED_FONT	Font với kích thước cố định của ký tự dựa trên Windows. Font **Courier** là một ví dụ điển hình của dạng font này.
ANSI_VAR_FONT	Font với độ rộng ký tự thay đổi dựa trên các ký tự chuẩn của Windows. Font **MS San Serif** là một ví dụ điển hình.
DEVICE_DEFAULT_FONT	Font với thiết bị đã cho được chọn mặc nhiên. Dạng font này thường có sẵn trong hệ thống để điều khiển việc trình bày trên thiết bị. Tuy nhiên, đối với một số thiết bị, font được cài đặt ngay trên thiết bị. Ví dụ, đối với máy in, các font thiết bị cài sẵn thực hiện thao tác in nhanh hơn so với việc load bitmap ảnh về

	từ máy tính.
DEFAULT_GUI_FONT	Font của giao diện đồ họa được thiết lập mặc định.
OEM_FIXED_FONT	Font chữ cố định, dựa trên bộ ký tự **OEM**. Ví dụ, đối với máy IBM®, font **OEM** dựa trên bộ ký tự IBM PC.
SYSTEM_FONT	Font hệ thống của Windows. Được hệ điều hành dùng để trình bày các thành phần giao diện như thanh tiêu đề, menu, nội dung văn bản trong các hộp thoại thông điệp. Các font hệ thống này luôn có sẵn khi cài hệ điều hành, trong khi các font khác cần phải cài thêm tùy theo ứng dụng sau này.
SYSTEM_FIXED_FONT	Font Windows được sử dụng như font hệ thống trong các phiên bản trước 3.0.

Bảng 5.1 Macro các font định nghĩa sẵn.

Việc chọn và sử dụng font hệ thống trong Windows khá đơn giản. Để làm điều này, đầu tiên chương trình tạo ra handle của font - kiểu biến HFONT, sau đó chọn font dùng hàm **GetStockObject**.

```
HGDIOBJ GetStockObject(int fnObject);
```

Trong đó, kiểu **HGDIOBJ** là **HFONT**, biến fnObject là một trong các macro ở bảng trên. Nếu thành công, hàm này trả về handle của font hệ thống hiện hành. Ngược lại, giá trị trả về là NULL. Để thay đổi font, ta gọi hàm **SelectObject**.

```
HGDIOBJ SelectObject(HDC hDC, HGDIOBJ hGDIObj);
```

Hoặc gọn hơn, ta có thể gọi:

```
SelectObject(hDC, GetStockObject(fnObject));
```

Khi đó, font hiện hành trong DC là font vừa được gọi. Hàm trả về macro font trước đó. Nếu không thành công, lỗi trả về là GDI_ERROR. Thường khi gọi font định sẵn, nếu các font không có sẵn, hệ thống sẽ trả về font hệ thống (SYSTEM_FONT). Lưu ý, chỉ nên dùng các font có sẵn nếu chế độ hiện thị của DC ứng dụng hiện thời là MM_TEXT.

Để xem các thuộc tính của font hệ thống, ví dụ kích thước của bộ font để tính toán vị trí khi xuất văn bản, ta dùng hàm **GetTextMetrics**.

```
BOOL GetTextMetrics(HDC hDC, LPTEXTMETRIC lpTM);
```

Biến lpTM là con trỏ đến cấu trúc **TEXTMETRIC** mà nếu hàm thực hiện thành công (trả về giá trị nonzero) sẽ chứa các tham số của font.

Đoạn chương trình sau minh họa việc chọn font định sẵn vào một DC, sau đó viết một chuỗi ký tự sử dụng font này.

```
HFONT hfnt, hOldFont;
hfnt = GetStockObject(ANSI_VAR_FONT);
if (hOldFont = SelectObject(hdc, hfnt)) {
   TextOut(hdc, 10, 50, "Sample ANSI_VAR_FONT text.", 26);
   SelectObject(hdc, hOldFont);
}
```

Tuy nhiên, font chữ hệ thống khá nghèo nàn, không đáp ứng đủ nhu cầu trình bày văn bản. Vì thế, Windows hỗ trợ các hàm sử dụng các font chữ tự tạo. Phần tiếp theo sẽ bàn kỹ hơn về các hàm xử lý các font này.

3. Sử dụng font tự tạo

Font logic là một đối tượng **GDI** có handle kiểu **HFONT**. Một font logic là một mô tả của một font chữ thực tế. Tương tự như viết vẽ logic và chổi sơn logic, font logic là một đối tượng trừu tượng, và trở thành font thực tế trong DC khi ứng dụng gọi hàm **SelectObject**.

Đầu tiên, ta cần định nghĩa các trường trong cấu trúc **LOGFONT** theo một trong hai cách sau:

- Thiết lập các trường trong cấu trúc **LOGFONT** dựa trên các đặc tính của font bạn quan tâm. Trong trường hợp này, khi gọi hàm **SelectObject**, Windows sẽ sử dụng một thuật toán "ánh xạ font" để chọn một font tương thích nhất hiện có trên thiết bị.
- Chọn font trong hộp thoại Font, hàm **ChooseFont** sẽ khởi gán các giá trị trong cấu trúc **LOGFONT** theo thuộc tính font được chọn.

Để sử dụng font này trong các thao tác xuất văn bản, trước hết ứng dụng cần khởi tạo font logic, rồi chuyển cho DC. Việc tạo font logic được thực hiện bằng cách gọi hàm **CreateFont** hoặc hàm **CreateFontIndirect**. Tuy nhiên, hàm **CreateFont** ít được sử dụng hơn do hàm cần gởi các biến trong cấu trúc **LOGFONT**, trong khi hàm **CreateFontIndirect** chỉ cần biến con trỏ **LOGFONT**. Cả hai hàm đều trả về handle của font logic **HFONT**.

Trước khi ứng dụng có thể xuất văn bản, font logic cần được ánh xạ tương ứng với một font vật lý trong thiết bị xuất, hoặc font nạp vào hệ điều hành. Tiến trình này được thực hiện khi ứng dụng gọi hàm

SelectObject. Khi đó, ta có thể dùng hàm **GetTextMetrics** (hoặc các hàm tương đương) xác định kích thước và đặc trưng của font thực để tính vị trí và xuất văn bản ra thiết bị xuất.

Sau khi dùng font, ta cần loại bỏ font dùng hàm **DeleteObject**. Chú ý không được xoá font trong DC đang thực thi, hoặc font hệ thống.

4. Cấ u trúc LOGFONT và TEXTMETRIC

Để tạo font logic, ta có thể gọi hàm **CreateFont** hoặc **CreateFontIndirect**. Hàm **CreateFontIndirect** nhận con trỏ cấu trúc **LOGFONT**. Hàm **CreateFont** có biến là 14 trường của **LOGFONT**. Xét định nghĩa của cấu trúc **LOGFONT:**

```
typedef struct tagLOGFONT {
   LONG lfHeight;
   LONG lfWidth;
   LONG lfEscapement;
   LONG lfOrientation;
   LONG lfWeight;
   BYTE lfItalic;
   BYTE lfUnderline;
   BYTE lfStrikeOut;
   BYTE lfCharSet;
   BYTE lfOutPrecision;
   BYTE lfClipPrecision;
   BYTE lfQuality;
   BYTE lfPitchAndFamily;
   TCHAR lfFaceName[LF_FACESIZE];
} LOGFONT;
```

Hai trường đầu trong cấu trúc **LOGFONT** tính theo đơn vị logic, do đó phụ thuộc vào chế độ hiển thị. Trường `lfHeight` là chiều cao của ký tự, nhận giá trị mặc định của font nếu bằng 0. Nếu `lfHeight` âm, Windows xác định kích cỡ font qua giá trị tuyệt đối gần nhất. Giá trị dương của `lfHeight` xấp xỉ bằng giá trị `tmHeight` tương ứng trong cấu trúc **TEXTMETRICS**. Trường `lfWidth` xác định độ rộng ký tự. Trong trường hợp `lfWidth` bằng 0, Windows sẽ chọn font tương ứng dựa vào chiều cao `lfHeight`. Giá trị `lfWidth` khác 0 thường được sử dụng cho các font **TrueType** (không tương thích cho các font vector). Trường này tương đương trường `tmAveCharWidth` trong cấu trúc **TEXTMETRICS**.

Hai trường tiếp theo xác định hướng hiển thị của các ký tự trong chuỗi. Trường `lfEscapement` xác định góc lệch giữa đường cơ sở (*base line*) và trục **x** của thiết bị. Giá trị nhập vào là một bội số của 10. Chuỗi xuất theo hướng mặc định (trái sang phải), hướng lên, từ phải sang trái, và hướng xuống nếu giá trị lần lượt là 0, 900, 1800, và 2700. Trường lfOrientation xác định hướng đường cơ sở của từng ký tự so với trục x. Giá trị được xác định tương tự `lfEscapement`. Nếu ứng dụng gọi hàm trong Windows NT, với chế độ đồ họa là `GM_ADVANCED`, giá trị `lfEscapement` được xác định độc lập với `lfOrientation`. Trong các trường hợp khác, hai giá trị này phải được thiết lập như nhau.

Ví dụ sau thể hiện việc xuất một chuỗi quanh tâm vùng làm việc của cửa sổ ứng dụng bằng cách thay đổi giá trị của trường `lfEscapement` và `lfOrientation` từng 10 độ.

```
RECT rc;
int angle;
HFONT hfnt, hfntPrev;
LPSTR lpszRotate = "String to be rotated.";
// Cấp phát vùng nhớ cho cấu trúc LOGFONT.
PLOGFONT plf=(PLOGFONT)LocalAlloc(LPTR, sizeof(LOGFONT));
// Xác định tên và độ đậm nhạt của font.
lstrcpy(plf->lfFaceName, "Arial");
plf->lfWeight = FW_NORMAL;
// Thu nhận kích thước của vùng làm việc cửa sổ ứng dụng.
GetClientRect(hwnd, &rc);
// Thiết lập chế độ nền TRANSPARENT cho thao tác xuất văn bản.
SetBkMode(hdc, TRANSPARENT);
// Vẽ chuỗi quay quanh tâm vùng làm việc từng 10 độ (36 lần).
for (angle = 0; angle < 3600; angle += 100) {
   plf->lfEscapement = angle;
   hfnt = CreateFontIndirect(plf);
```

```
        hfntPrev = SelectObject(hdc, hfnt);
        TextOut(hdc, rc.right/2, rc.bottom/2, lpszRotate, lstrlen(lpszRotate));
        SelectObject(hdc, hfntPrev);
        DeleteObject(hfnt);
    }
    // Chuyển chế độ hiển thị về lại dạng OPAQUE.
    SetBkMode(hdc, OPAQUE);
    // Giải phóng vùng nhớ đã cấp phát cho cấu trúc LOGFONT.
    LocalFree((LOCALHANDLE) plf);
```

Trường tiếp theo là `lfWeight` xác định độ độ đậm nhạt của ký tự, có giá trị từ 0 đến 1000. Trong trường hợp bằng 0, Windows sử dụng giá trị mặc định. Để tiện dùng, ta dùng một số macro sau:

Tên	Giá trị
FW_DONTCARE	0
FW_THIN	100
FW_EXTRALIGHT	200
FW_ULTRALIGHT	200
FW_LIGHT	300
FW_NORMAL	400
FW_REGULAR	400
FW_MEDIUM	500
FW_SEMIBOLD	600
FW_DEMIBOLD	600
FW_BOLD	700
FW_EXTRABOLD	800
FW_ULTRABOLD	800
FW_HEAVY	900
FW_BLACK	900

Bảng 5.2 Macro xác định độ đậm nhạt lfWeight.

Các trường `lfItalic`, `lfUnderline`, `lfStrikeOut` xác định font nghiêng, có gạch dưới, gạch ngang nếu được thiết lập bằng `TRUE`.

Trường `lfCharSet` xác định tập ký tự sử dụng. Giá trị `OEM_CHARSET` xác định tập ký tự phụ thuộc hệ điều hành. Thông thường, ta sử dụng `DEFAULT_CHARSET` (tập ký tự mặc định). Việc chọn tập ký tự là rất quan trọng, nếu ứng dụng sử dụng font có tập ký tự không xác định, hệ thống không thể thông dịch chúng.

Các trường còn lại khá phức tạp. Chúng ta không phân tích các giá trị của chúng ở đây. Trường `lfOutPrecision` định nghĩa độ chính xác (có thể có) của font thực tế so với font yêu cầu. Trường `lfClipPrecision` xác định cách thức các ký tự bị cắt bỏ khi nằm ngoài vùng clipping. Trường `lfQuality` dùng cho font vector, xác định cách thức GDI định nghĩa độ chính xác của font logic so với font vật lý. Trường `lfPitchAndFamily` xác định họ font chữ. Trường `lfFaceName` xác định tên kiểu font.

Sau khi đã chọn font cho DC, ta cần biết các đặc trưng của font hiện tại dùng cho việc xuất văn bản bằng hàm **GetTextMetrics**. Ta có cấu trúc **TEXTMETRIC:**

```
    typedef struct tagTEXTMETRIC {
        LONG tmHeight;
        LONG tmAscent;
        LONG tmDescent;
        LONG tmInternalLeading;
        LONG tmExternalLeading;
        LONG tmAveCharWidth;
        LONG tmMaxCharWidth;
        LONG tmWeight;
        LONG tmOverhang;
        LONG tmDigitizedAspectX;
        LONG tmDigitizedAspectY;
        BCHAR tmFirstChar;
        BCHAR tmLastChar;
        BCHAR tmDefaultChar;
```

```
    BCHAR tmBreakChar;
    BYTE tmItalic;
    BYTE tmUnderlined;
    BYTE tmStruckOut;
    BYTE tmPitchAndFamily;
    BYTE tmCharSet;
} TEXTMETRIC;
```

Các trường về kích thước được tính theo đơn vị logic, phụ thuộc vào chế độ hiển thị. Trường tmHeight cho biết chiều cao của ký tự, bằng tổng hai trường tmAscent (phần trên đường cơ sở) và tmDescent (phần dưới đường cơ sở). Trường tmInternalLeading xác định khoảng cách bên trong đường biên xác định bởi tmHeight. Người thiết kế có thể cho giá trị này bằng 0. Trường tmExternalLeading là khoảng cách Windows thêm vào giữa các dòng. Người dùng cũng có thể gán giá trị này bằng 0. Trường tmAveCharWidth cho biết độ rộng trung bình của các ký tự chữ thường. Trường tmMaxCharWidth cho biết độ rộng ký tự lớn nhất tính theo đơn vị logic. Đối với bộ font fixed-pitch (độ rộng không đổi), giá trị này bằng tmAveCharWidth.

Trường tmOverhang cho biết phần mở rộng (theo đơn vị logic) mà Windows thêm vào font vector khi font nghiêng hoặc tô đậm. Khi font nghiêng, giá trị tmAveCharWidth không thay đổi, bởi vì chuỗi các ký tự nghiêng vẫn có độ rộng bằng chuỗi chữ thông thường. Trong trường hợp chữ đậm, Windows sẽ mở rộng từng ký tự. Khi đó, tmAveCharWidth của ký tự bình thường bằng tmAveCharWidth trừ đi độ lớn tmOverhang của ký tự đậm.

Trường tmDigitizedAspectX và tmDigitizedAspectY xác định tỉ lệ co của font. Chúng ta cũng có thể xác định chúng bằng hàm **GetDeviceCaps** (xác định các thông tin thiết bị) với các thông số của nIndex là **LOGPIXELSX** và **LOGPIXELSY**.

```
int GetDeviceCaps(HDC hDC, int nIndex);
```

Trường tmFirstChar xác định giá trị ký tự đầu tiên trong font, tmLastChar xác định giá trị ký tự cuối cùng. Nếu gọi hàm **GetTextMetricsW**, giá trị này sẽ lớn hơn 255. Trường tmDefaultChar chứa giá trị Windows dùng để hiển thị các ký tự không tồn tại trong bộ font, thường là hình chữ nhật. Trường tmBreakChar được dùng để ngắt từ khi canh chỉnh văn bản. Nếu không dùng các font đặc biệt, ví dụ font **EBCDIC**, giá trị này chính là giá trị ký tự khoảng trắng (mã bằng 32).

Các trường tmWeight, tmItalic, tmUnderlined, tmStruckOut, tmCharSet và tmPitchAndFamily giống các trường lfWeight, lfItalic, lfUnderline, lfStrikeOut, lfCharSet và lfPitchAndFamily trong cấu trúc **LOGFONT**.

5. Sử dụng hộp thoại chọn font

Để chọn font, ta có thể sử dụng hộp thoại font. Đầu tiên, ứng dụng khởi tạo các giá trị trong cấu trúc **CHOOSEFONT**, sau đó gọi hàm **ChooseFont** để hiển thị hộp thoại liệt kê danh sách các font. Sau khi người dùng chọn một font có sẵn và chọn **OK**, hàm **ChooseFont** sẽ khởi gán các giá trị tương ứng vào cấu trúc **LOGFONT**. Ứng dụng có thể gọi hàm **CreateFontIndirect** để tạo một font logic dựa trên các yêu cầu của người dùng.

Xét cấu trúc **CHOOSEFONT**.

```
typedef struct tagCHOOSEFONT {
    DWORD lStructSize;
    HWND hwndOwner;
    HDC hDC;
    LPLOGFONT lpLogFont;
    INT iPointSize;
    DWORD Flags;
    DWORD rgbColors;
    LPARAM lCustData;
    LPCFHOOKPROC lpfnHook;
    LPCTSTR lpTemplateName;
    HINSTANCE hInstance;
    LPTSTR lpszStyle;
    WORD nFontType;
    WORD ___MISSING_ALIGNMENT__;
    INT nSizeMin;
    INT nSizeMax;
} CHOOSEFONT;
```

Trường lStructureSize xác định số byte kích thước cấu trúc. Trường hwndOwner xác định cửa sổ chứa hộp thoại chọn font. Trường này có thể là một handle cửa sổ xác định, hoặc cũng có thể bằng NULL nếu hộp thoại không xác định cửa sổ cha. Trường hDC là DC của máy in có các font được liệt kê trong hộp thoại. Chỉ dùng trường hDC nếu thiết lập trường Flags với cờ CF_PRINTERFONTS hoặc CF_BOTH; trong các trường hợp khác, giá trị trường này bị bỏ qua.

Trường lpLogFont trỏ đến cấu trúc **LOGFONT**. Nếu ta thiết lập trong trường Flags giá trị CF_INITTOLOGFONTSTRUCT, và khởi gán các giá trị cho nó, hàm **ChooseFont** sẽ khởi tạo hộp thoại với font gần giống nhất. Sau khi ta chọn **OK** đóng hộp thoại, hàm **ChooseFont** sẽ thiết lập các giá trị của cấu trúc này dựa trên các chọn lựa tương ứng.

Trường iPointSize xác định kích thước font được chọn theo đơn vị 1/10 điểm. Giá trị này sẽ được thiết lập sau khi ta đóng hộp thoại.

Trường Flags là tập các bit cờ được dùng để khởi tạo hộp thoại Font. Trường này có thể kết hợp các giá trị sau:

CF_APPLY	Hộp thoại có thêm button **Apply**. Ta phải cung cấp hàm hook xử lý thông điệp WM_COMMAND cho button này.
CF_BOTH	Hộp thoại liệt kê danh sách các font máy in và font màn hình. Trường hDC sẽ xác định DC liên quan đến máy in. Cờ này cần kết hợp với cờ CF_SCREENFONTS và CF_PRINTERFONTS.
CF_TTONLY	Xác định hàm **ChooseFont** chỉ quản lý và chọn lựa trên font dạng **TrueType**.
CF_EFFECT	Để hộp thoại thể hiện các control cho phép người dùng thiết lập chữ gạch ngang, gạch dưới và chọn màu. Nếu cờ này được thiết lập, ta có thể sử dụng trường rgbColors xác định màu font ban đầu. Ta cũng có thể sử dụng trường lfStrikeOut và lfUnderline của cấu trúc **LOGFONT** trỏ đến trường lpLogFont xác định các thiết lập ban đầu của checkbox gạch ngang và gạch dưới ký tự. Hàm **ChooseFont** sẽ sử dụng các trường này để trả về giá trị chọn lựa của người dùng.
CF_ENABLEHOOK	Cho phép thủ tục hook thiết lập trong trường lpfnHook.
CF_ENABLETEMPLATE	Chỉ ra trường hInstance và lpTemplateName xác định hộp thoại mẫu sử dụng mặc định.
CF_ENABLETEMPLATEHANDLE	Chỉ ra trường hInstance xác định khối dữ liệu chứa hộp thoại mẫu trước khi nạp. Hệ thống sẽ bỏ qua giá trị của trường lpTemplateName.
CF_FORCEFONTEXIST	Xác định hàm **ChooseFont** sẽ chỉ ra lỗi nếu người dùng cố chọn font hoặc kiểu không tồn tại.
CF_INITTOLOGFONTSTRUCT	Xác định hàm **ChooseFont** phải sử dụng cấu trúc **LOGFONT** qua trường lpLogFont để khởi tạo hộp thoại.
CF_LIMITSIZE	Xác định hàm **ChooseFont** chỉ chọn font có kích thước trong giới hạn từ nSizeMin đến nSizeMax.
CF_NOOEMFONTS	Giống cờ CF_NOVECTORFONTS.
CF_NOFACESEL	Khi sử dụng cấu trúc **LOGFONT** để khởi tạo các control của hộp thoại, trong đó combo box tên font không thể hiện một font nào đang được chọn cả.
CF_NOSCRIPTSEL	Không cho phép chọn combo box **Script**. Khi cờ này được thiết lập, trường lfCharSet của cấu trúc **LOGFONT** được thiết lập với giá trị DEFAULT_CHARSET khi hàm **ChooseFont** trả về. Cờ này chỉ dùng khi khởi tạo hộp thoại.
CF_NOSTYLESEL	Khi sử dụng cấu trúc **LOGFONT** để khởi tạo các control của hộp thoại, ta không dùng kiểu font nào cả (*combo box style*).
CF_NOSIZESEL	Khi sử dụng cấu trúc **LOGFONT** để khởi tạo các control của hộp thoại, ta không hiển thị chọn lựa ban đầu cho combo box kích thước.
CF_NOVECTORFONTS	Hàm **ChooseFont** không cho phép chọn lựa font vector.
CF_NOVERTFONTS	Hộp thoại font chỉ liệt kê các font viết hướng ngang.
CF_PRINTERFONTS	Hộp thoại chỉ liệt kê các font hỗ trợ máy in liên quan đến DC xác định qua handle hDC.
CF_SCALABLEONLY	Chỉ định hàm **ChooseFont** chỉ cho phép chọn lựa các font có thể co giãn (font vector, TrueType, …).

CF_SCREENFONTS	Hộp thoại chỉ liệt kê các font màn hình được hệ thống hỗ trợ.
CF_SHOWHELP	Hộp thoại sẽ hiển thị nút **Help**. Trường hwndOwner phải xác định window nhận thông điệp **HELPMSGSTRING** mà hộp thoại đưa ra khi người dùng nhấn button **Help**.
CF_USESTYLE	Xác định trường lpszStyle trỏ đến vùng đệm chứa dữ liệu kiệu mà hàm **ChooseFont** dùng để khởi tạo combo box **Font Style**. Sau khi người dùng đóng hộp thoại, hàm **ChooseFont** chép dữ liệu kiểu vào vùng đệm.
CF_WYSIWYG	Xác định hàm **ChooseFont** cho phép chọn lựa các font có sẵn cho máy in và màn hình. Nếu cờ này được thiết lập, cả cờ CF_BOTH và CF_SCALABLEONLY cũng cần được thiết lập.

Bảng 5.3 Các macro cờ Flags khởi tạo hộp thoại Font.

Nếu cờ CF_EFFECT được thiết lập, trường rgbColors sẽ xác định màu chữ khởi tạo. Nếu hàm **ChooseFont** thực hiện thành công, trường này sẽ chứa giá trị **RGB** của màu chữ mà người dùng đã chọn.

Trường lCustData xác định dữ liệu được ứng dụng định nghĩa chuyển cho hàm hook gán bởi giá trị lpfnHook. Nếu hệ thống gởi thông điệp **WM_INITDIALOG**, giá trị **lParam** của thông điệp là con trỏ trỏ đến cấu trúc **CHOOSEFONT** xác định khi hộp thoại được tạo.

Trường lpfnHook trỏ đến thủ tục hook CFHookProc xử lý các thông điệp của hộp thoại. Để sử dụng giá trị này, cần phải thiết lập cờ với CF_ENABLEHOOK.

Trường lpTemplateName trỏ đến chuỗi (kết thúc bằng ký tự NULL) xác định tên tài nguyên hộp thoại mẫu, liên quan với trường hInstance. Nếu thiết lập cờ CF_ENABLETEMPLATEHANDLE, hInstance là handle của đối tượng vùng nhớ chứa hộp thoại mẫu. Nếu thiết lập cờ CF_ENABLETEMPLATE, hInstance xác định module chứa hộp thoại mẫu có tên ở trường lpTemplateName.

Trường lpszStyle trỏ đến vùng đệm chứa dữ liệu kiểu. Nếu thiết lập cờ CF_USESTYLE, hàm **ChooseFont** sẽ sử dụng dữ liệu trong vùng đệm để khởi tạo combo box kiểu font. Khi người dùng đóng hộp thoại, hàm **ChooseFont** chép chuỗi trong combo box kiểu font vào vùng đệm này.

Trường nFontType xác định dạng font trả về. Có thể kết hợp các giá trị sau:

BOLD_FONTTYPE	Font chữ đậm. Giá trị này được lặp lại trong trường lfWeight của cấu trúc **LOGFONT**, và bằng FW_BOLD.
ITALIC_FONTTYPE	Font chữ nghiêng. Giá trị này được lặp lại trong trường lfItalic của cấu trúc **LOGFONT**.
PRINTER_FONTTYPE	Font chữ là font máy in.
REGULAR_FONTTYPE	Font chữ bình thường. Giá trị này được lặp lại trong trường lfWeight của cấu trúc **LOGFONT**, và bằng FW_REGULAR.
SCREEN_FONTTYPE	Font chữ là font màn hình.
SIMULATED_FONTTYPE	Font dựa theo giao diện GUI.

Bảng 5.4 Các macro xác định dạng font trả về.

Trường nSizeMin và nSizeMax xác định kích cỡ chữ nhỏ nhất và lớn nhất mà người dùng có thể chọn. Hàm **ChooseFont** chỉ nhận ra các giá trị này khi cờ CF_LIMITSIZE được thiết lập.

Như đã trình bày ở trên, để tạo hộp thoại font cho phép người dùng chọn các thuộc tính của một font logic, ta dùng hàm **ChooseFont**.

```
BOOL ChooseFont(LPCHOOSEFONT lpcf);
```

Với lpcf trỏ đến cấu trúc **CHOOSEFONT** chứa các thông tin dùng để khởi tạo hộp thoại. Khi hàm trả về, cấu trúc này chứa các thông tin mà người dùng chọn.

Nếu người dùng nhấn nút **OK** trên hộp thoại, hàm trả về giá trị khác 0. Ngược lại, nếu chọn **Cancel** hoặc đóng (*close*) hộp thoại, giá trị trả về bằng 0.

Ví dụ sau minh họa việc chọn font logic sử dụng hộp thoại font.

```
HFONT FAR PASCAL MyCreateFont(void) {
   CHOOSEFONT cf;
   LOGFONT lf;
   HFONT hfont;
   // Khởi tạo các trường trong cấu trúc CHOOSEFONT.
   cf.lStructSize = sizeof(CHOOSEFONT);
   cf.hwndOwner = (HWND)NULL;
```

```
        cf.hDC = (HDC)NULL;
        cf.lpLogFont = &lf;
        cf.iPointSize = 0;
        cf.Flags = CF_SCREENFONTS;
        cf.rgbColors = RGB(0,0,0);
        cf.lCustData = 0L;
        cf.lpfnHook = (LPCFHOOKPROC)NULL;
        cf.lpTemplateName = (LPSTR)NULL;
        cf.hInstance = (HINSTANCE) NULL;
        cf.lpszStyle = (LPSTR)NULL;
        cf.nFontType = SCREEN_FONTTYPE;
        cf.nSizeMin = 0;
        cf.nSizeMax = 0;
        // Hiển thị hộp thoại font.
        ChooseFont(&cf);
        /* Tạo font logic dựa trên các chọn lựa của người dùng và trả về handle
           của font logic */
        hfont = CreateFontIndirect(cf.lpLogFont);
        return (hfont);
    }
```

6. Ví dụ về hiển thị văn bản bằng các font khác nhau

Sau khi tìm hiểu về việc xuất nội dung văn bản và sử dụng font chữ bằng các hàm Win32® API, ta có thể trình bày văn bản một cách linh động hơn. Phần này đưa ra ví dụ xuất một chuỗi văn bản theo các font khác nhau trên cùng một dòng.

Một trong các cách trình bày một dòng văn bản với nhiều font chữ khác nhau là sử dụng hàm **GetTextExtentPoint32** sau mỗi lần gọi hàm **TextOut** và cộng chiều dài nhận được vào vị trí hiển thị hiện tại. Ví dụ sau xuất câu "**This is a** *sample* **string**", sử dụng chữ đậm để viết "**This is a**", chữ nghiêng để viết "*sample*", và trở lại chữ đậm để viết "**string**.". Sau khi xuất xong, chương trình phục hồi lại dạng font mặc định.

```
    int XIncrement;
    int YStart;
    TEXTMETRIC tm;
    HFONT hfntDefault, hfntItalic, hfntBold;
    SIZE sz;
    LPSTR lpszString1 = "This is a ";
    LPSTR lpszString2 = "sample ";
    LPSTR lpszString3 = "string.";
    /* Tạo font chữ đậm và nghiêng (hàm này không viết ở đây) */
    hfntItalic = MyCreateFont();
    hfntBold = MyCreateFont();
    /* Chọn font chữ đậm và viết chuỗi tại vị trí xác định bởi cặp tham số
       (XIncrement, YStart) */
    XIncrement = 10;
    YStart = 50;
    hfntDefault = SelectObject(hdc, hfntBold);
    TextOut(hdc, XIncrement, YStart, lpszString1, lstrlen(lpszString1));
    /* Tính độ dài của chuỗi đầu tiên và cộng giá trị này vào giá trị Xincrement
       để thực hiện thao tác xuất chuỗi tiếp theo */
    GetTextExtentPoint32(hdc, lpszString1, lstrlen(lpszString1), &sz);
    XIncrement += sz.cx;
    /* Xác định độ lớn phần mở rộng ký tự qua cấu trúc TEXTMETRIC và trừ bớt
       XIncrement. (Điều này chỉ cần thiết cho font vector) */
    GetTextMetrics(hdc, &tm);
    XIncrement -= tm.tmOverhang;
    /* Chọn font nghiêng và viết chuỗi thứ hai bắt đầu tại vị trí
       (XIncrement, YStart) */
    hfntBold = SelectObject(hdc, hfntItalic);
    GetTextMetrics(hdc, &tm);
    XIncrement -= tm.tmOverhang;
```

```
TextOut(hdc, XIncrement, YStart, lpszString2, lstrlen(lpszString2));
/* Tính độ dài của chuỗi đầu tiên và cộng giá trị này vào giá trị Xincrement
   để thực hiện thao tác xuất chuỗi tiếp theo */
GetTextExtentPoint32(hdc, lpszString2, lstrlen(lpszString2), &sz);
XIncrement += sz.cx;
/* Chọn lại font chữ đậm và viết chuỗi thứ ba bắt đầu tại vị trí
   (XIncrement, YStart) */
SelectObject(hdc, hfntBold);
TextOut(hdc, Xincrement - tm.tmOverhang, YStart,
   lpszString3, lstrlen(lpszString3));
/* Chọn lại font ban đầu */
SelectObject(hdc, hfntDefault);
/* Xoá các đối tượng font chữ */
DeleteObject(hfntItalic);
DeleteObject(hfntBold);
```

Trong ví dụ trên, ta sử dụng hàm **GetTextExtentPoint32** để nhận chiều dài và chiều cao của chuỗi văn bản.

```
BOOL GetTextExtentPoint32(HDC hDC, LPCTSTR lpString,
                          int cbString, LPSIZE lpSize);
```

Trong đó `hDC` là handle của DC, `lpString` trỏ đến chuỗi văn bản có chiều dài xác định bởi `cbString`. Và `lpSize` trỏ đến cấu trúc **SIZE** chứa kích thước của chuỗi ký tự cần xác định.

```
typedef struct tagSIZE {
   LONG cx;
   LONG cy;
} SIZE;
```

Cấu trúc trên xác định chiều rộng và chiều cao của một hình chữ nhật lần lượt qua hai biến `cx` và `cy`.

Nếu thực hiện thành công, hàm trả về giá trị khác 0. Ngược lại, giá trị trả về bằng 0.

CHƯƠNG VI: ĐỒ HỌA VÀ CÁC ĐỐI TƯỢNG GDI

A. MỞ ĐẦU

Windows cung cấp một tính năng rất đặc sắc, đó là khả năng đồ họa độc lập thiết bị được xây dựng trên kỹ thuật **GDI** (giao diện giao tiếp với các thiết bị đồ họa khác nhau). GDI là thư viện đồ họa của Windows, cung cấp tất cả hàm phục vụ cho các thao tác kết xuất hình ảnh và văn bản ra thiết bị.

GDI có thể vẽ ra nhiều loại thiết bị khác nhau:

- **Màn hình**
- **Máy in**
- **Máy vẽ**

GDI có trách nhiệm giao tiếp và kết xuất các yêu cầu mà người dùng chuyển cho nó đến đúng thiết bị đích. Về cơ bản, nó giao tiếp với các trình điều khiển thiết bị (các tập tin.**drv**), thật ra các trình điều khiển thiết bị cũng là một giao diện do Windows đưa ra, do đó trách nhiệm nặng nề không thật sự thuộc về GDI của Windows mà là của các nhà sản xuất thiết bị phần cứng, họ buộc phải cung cấp trình điều khiển theo giao diện này nếu muốn bán được sản phẩm cho người dùng Windows. Như vậy, người lập trình không cần quan tâm đến việc điều khiển trực tiếp thiết bị xuất mà chỉ cần quan tâm đến thư viện hàm GDI.

Chương này sẽ trình bày các khái niệm cơ sở về GDI như device context, các hàm GDI cơ sở để vẽ và tô, các hàm để nạp và zoom ảnh bitmap. Phần cuối chương sẽ trình bày cách lấy về handle device context của máy in và một số hàm cơ sở sử dụng cho việc điều khiển in ấn.

Tóm lại, Windows cung cấp khả năng sử dụng cùng một hàm để kết xuất ra nhiều thiết bị khác nhau. Điều này làm cho chương trình độc lập với thiết bị.

B. DEVICE CONTEXT

Device context là một thiết bị xuất logic, liên kết với một thiết bị xuất vật lý cụ thể. Windows không cho phép chúng ta kết xuất trực tiếp ra thiết bị vật lý mà phải thông qua handle của device context. Handle device context là một số nguyên không dấu được Windows cấp như một định danh của device context.

Ví dụ 1: Xuất dòng chữ "Hello Windows 2000" ra màn hình

```
HDC hDC;
/* Lấy device context của cửa sổ */
hDC = GetDC(hWnd);
/* Xuất dòng chữ "Hello Windows 2000" ra cửa sổ tại vị trí (20,20) */
TextOut(hDC, 20, 20, "Hello Windows 2000", 18);
/*Giải phóng Device Context */
RealeaseDC(hWnd, hDC);
```

Ví dụ 2: Vẽ hình chữ nhật

```
HDC hDC;
HPEN hPen, oldHPen;
/* Lấy về device context của cửa sổ */
hDC = GetDC(hWnd);
/* Tạo bút vẽ mới với nét liền, độ dày 5, màu xanh */
hPen = CreatePen(PS_SOLID, 5, RGB(0, 0, 255));
/* Chọn bút vẽ hiện hành là bút vẽ mới và lưu lại bút vẽ cũ */
oldHPen = (HPEN)SelectObject(hDC, hPen);
/* Vẽ hình chữ nhật */
Rectangle(hDC, 20, 20, 100, 100);
/* Restore lại bút vẽ cũ */
SelectObject(hDC, oldHPen);
/* Giải phóng bút vẽ đã tạo ra */
DeleteObject(hPen);
/* Giải phóng device context */
ReleaseDC(hWnd, hDC);
```

1. Thao tác lấy về và giải phóng Device Context

Có 3 cách để nhận về và giải phóng một Device Context:

Sử dụng hàm **BeginPaint** và **EndPaint** khi xử lý thông điệp **WM_PAINT**:

```
hDC = BeginPaint(hWnd, &ps);
// Xử lý
EndPaint(hWnd, &ps);
```

Biến ps là một cấu trúc kiểu **PAINTSTRUCT**.

Dùng hàm **GetDC** và **ReleaseDC** khi xử lý các thông điệp khác **WM_PAINT**:

```
hDC = GetDC(hWnd);
//Xử lý
ReleaseDC(hWnd, hDC);
```

Dùng hàm **GetWindowDC** và **ReleaseDC:**

```
hDC = GetWindowDC(hWnd);
// Xử lý
ReleaseDC(hWnd, hDC);
```

Lưu ý: Các hàm **GetDC** và **BeginPaint** trả về device context cho vùng client của cửa sổ, riêng hàm **GetWindowDC** trả về device context của toàn bộ cửa sổ kể cả thanh tiêu đề, menu, thanh cuộn… và tất nhiên là cả vùng client. Để vẽ ra ngoài vùng làm việc (*client area*), phải chặn thông điệp **WM_NCPAINT** (**NC**: *Non-Client*). Ngoài ra, còn có thể nhận về device context của toàn màn hình bằng hàm:

```
hDC = CreateDC("DISPLAY", NULL, NULL, NULL);
```

2. Tạo lập và giải phóng memory device context

Memory device context (*MDC*)là một device context ảo không gắn với một thiết bị xuất cụ thể nào. Muốn kết quả kết xuất ra thiết bị vật lý ta phải chép MDC lên một device context thật sự (device context có liên kết với thiết bị vật lý). MDC thường được dùng như một device context trung gian để vẽ trước khi thực sự xuất ra thiết bị, nhằm giảm sự chớp giật nếu thiết bị xuất là window hay màn hình.

Để tạo **MDC** tương thích với một **hDC** cụ thể, sử dụng hàm **CreateCompatibleDC**:

```
HDC hMemDC;
hMemDC = CreateCompatibleDC(hDC);
```

Đơn giản hơn, có thể đặt NULL vào vị trí hDC, Windows sẽ tạo một device context tương thích với màn hình.

Hủy MDC bằng hàm **DeleteDC**.

MDC có bề mặt hiển thị như một thiết bị thật. Tuy nhiên, bề mặt hiển thị này lúc đầu rất nhỏ, chỉ là một pixel đơn sắc. Không thể làm gì với một bề mặt hiển thị chỉ gồm 1 bit như vậy. Do đó cần làm cho bề mặt hiển thị này rộng hơn bằng cách chọn một đối tượng bitmap GDI vào MDC:

```
SelectObject(hMemDC, hBitmap);
```

Chỉ có thể chọn đối tượng bitmap vào MDC, không thể chọn vào một device context cụ thể được.

Sau khi chọn một đối tượng bitmap cho MDC, có thể dùng MDC như một device context thật sự.

Sau khi được hoàn tất trong MDC, ảnh được đưa ra device context thật sự bằng hàm **BitBlt**:

```
BitBlt(hDC, xDest, yDest, nWidth, nHeight, hMemDC, xSource, ySource);
```

Ví dụ: Chuẩn bị ảnh trước khi đưa ra màn hình, tránh gây chớp màn hình trong thông điệp **WM_PAINT**.

```
case WM_PAINT:
   hdc = BeginPaint(hWnd, &ps);
   // Lấy về kích thước vùng client của cửa sổ hiện hành
   RECT rect;
   GetClientRect(hWnd, &rect);
   // Tạo MDC tương thích với DC của cửa sổ
   HDC hMemDC;
   hMemDC = CreateCompatibleDC(hdc);
   // Chọn một đối tượng bitmap để mở rộng vùng hiển thị cho MDC
   HBITMAP bitmap,oBitmap;
   bitmap = CreateCompatibleBitmap(hdc,rect.right,rect.bottom);
   oBitmap = (HBITMAP)SelectObject(hMemDC,bitmap);
   // Vẽ lại nền MDC
   FillRect(hMemDC, &rect, HBRUSH (GetBkColor(hMemDC)));
   // Xuất hình ảnh, text ra MDC
   SetPixel(hMemDC, 0, 0, RGB(255,0,0));
   MoveToEx(hMemDC, 50, 50, NULL);
   LineTo(hMemDC, 100, 100);
   Rectangle(hMemDC, 10, 10, 100, 100);
   TextOut(hMemDC, 15,15, "Testing MDC", 11);
```

```
        if (!BitBlt(hdc, 0, 0, rect.right, rect.bottom, hMemDC, 0, 0, SRCCOPY))
           MessageBox(hWnd, "Failed to transfer bit block", "Error",MB_OK);
        // Phục hồi lại bitmap cũ cho MDC
        SelectObject(hMemDC, oBitmap);
        // Giải phóng MDC, bitmap đã tạo
        DeleteDC(hMemDC);
        DeleteObject(bitmap);
        EndPaint(hWnd, &ps);
        break;
```

C. MỘT SỐ HÀM ĐỒ HỌA CƠ SỞ

Sử dụng các hàm đồ họa cơ sở, ta có thể trình bày các đối tượng văn bản, hình ảnh … trên ứng dụng. Gồm các nhóm hàm về văn bản (*text*), bút vẽ, miền tô, và ảnh bitmap.

Các hàm về văn bản đã được trình bày trong chương trước. Ở đây sẽ trình bày các nhóm hàm còn lại.

1. Nhóm hàm vẽ

```
COLORREF GetPixel(HDC hDC, int nXPos, int nYPos);
```

Lấy về giá trị màu tại vị trí (nXPos, nYPos) của hDC, trả về -1 nếu điểm này nằm ngoài vùng hiển thị.

```
COLORREF SetPixel(HDC hDC, int nXPos, int nYPos, COLORREF clrRef);
```

Vẽ một điểm màu clrRef tại vị trí (nXPos, nYPos) lên hDC. Giá trị trả về là màu của điểm (nXPos, nYPos) hoặc -1 nếu điểm này nằm ngoài vùng hiển thị.

```
DWORD MoveToEx(HDC hDC, int x, int y);
```

Di chuyển bút vẽ đến tọa độ (x, y) trên hDC. Giá trị trả về là tọa độ cũ của bút vẽ, x = LOWORD, y = HIWORD.

```
BOOL LineTo(HDC hDC, int xEnd, int yEnd);
```

Vẽ đoạn thẳng từ vị trí hiện hành đến vị trí (xEnd, yEnd) trên hDC. Hàm trả về TRUE nếu thành công, FALSE nếu thất bại.

```
BOOL Polyline(HDC hDC, const POINT FAR *lpPoints, int nPoints);
```

Vẽ đường gấp khúc lên hDC bằng các đoạn thẳng liên tiếp, số đỉnh là nPoints với tọa độ các đỉnh được xác định trong lpPoints. Hàm trả về TRUE nếu thành công, FALSE nếu thất bại.

```
BOOL Polygon(HDC hDC, const POINT FAR *lpPoints, int nPoints);
```

Vẽ đa giác có nPoints đỉnh, tọa độ các đỉnh được xác định bởi lpPoints. Hàm trả về TRUE nếu thành công, FALSE nếu thất bại.

```
BOOL Rectangle(HDC hDC, int left, int top, int right, int bottom);
```

Vẽ hình chữ nhật có tọa độ là left, top, right, bottom lên hDC.

```
HPEN CreatePen(int penStyle, int penWidth, COLORREF penColor);
```

Tạo bút vẽ có kiểu penStyle, độ dày nét vẽ là penWidth, màu penColor. Hàm trả về handle của bút vẽ nếu thành công và trả về NULL nếu thất bại. Các giá trị của penStyle như sau:

Giá trị	Giải thích
PS_SOLID	
PS_DASH	
PS_DOT	
PS_DASHDOT	
PS_DASHDOTDOT	
PS_NULL	Không hiển thị
PS_INSIDEFRAME	

Bảng 6.1 Các kiểu bút vẽ penStyle

Ví dụ: Tạo bút vẽ mới và dùng bút vẽ này vẽ một số đường cơ sở.

```
HDC hDC;
POINT PointArr[3];
HPEN hPen, hOldPen;
hDC = GetDC(hWnd);
PointArr[0].x = 50;
PointArr[0].y = 10;
PointArr[1].x = 250;
```

```
PointArr[1].y = 50;
PointArr[2].x = 125;
PointArr[2].y = 130;
// Vẽ một đường gấp khúc bằng Pen hiện hành
Polyline(hDC, PointArr, 3);
//Tạo Pen mới có nét liền, độ dày 1, màu xanh
hPen = (HPEN)CreatePen(PS_SOLID, 1, RGB(0, 0, 255));
//Chọn bút vẽ mới cho hDC của window
hOldPen = SelectObject(hDC, hPen);
//Vẽ đường thẳng với bút vẽ mới
MoveToEx(hDC, 100, 100, NULL);
LineTo(hDC, 200, 150);
//Trả lại bút vẽ cũ cho hDC
SelectObject(hDC, hOldPen);
//Xoá bút vẽ mới tạo và giải phóng hDC
DeleteObject(hPen);
ReleaseDC(hWnd, hDC);
```

2. Nhóm hàm miền

```
HBRUSH CreateSolidBrush(COLORREF cRef);
```

Tạo mẫu tô đặc với màu `cRef`.

```
HBRUSH CreateHatchBrush(int bStyle, COLORREF cRef);
```

Tạo mẫu tô dạng lưới kiểu bStyle với màu `cRef`.

Các kiểu `bStyle`:

HS_HORIZONTAL
HS_VERTICAL
HS_FDIAGONAL
HS_BDIAGONAL
HS_CROSS
HS_DIAGCROSS

```
BOOL FloodFill(HDC hDC, int xStart, int yStart, COLORREF cRef);
```

Tô màu một vùng kín, màu đường biên là cRef.

```
BOOL ExtFloodFill(HDC hDC, int xStart, int yStart,
                  COLORREF cRef, UINT fillStyle);
```

Tô màu một vùng kín, `fillStyle` quyết định cách tô:

- `FLOODFILLBORDER`: Tô màu vùng có màu đường biên là `cRef`.
- `FLOODFILLSURFACE`: Tô vùng có màu `cRef`.

Ví dụ: Sử dụng các mẫu có sẵn và tạo các mẫu tô mới để tô.

```
HDC hDC;
HPEN hPen;
HBRUSH hBrush, hOldBrush;
hDC = GetDC(hWnd);
//Vẽ hai hình chữ nhật với bút vẽ Black
hPen = (HPEN)GetStockObject(BLACK_PEN);
SelectObject(hDC, hPen);
Rectangle(hDC, 10, 10, 50, 50);
Rectangle(hDC, 100, 100, 200, 200);
// Dùng một trong các mẫu tô có sẵn để tô hình
hBrush = (HBRUSH)GetStockObject(GRAY_BRUSH);
SelectObject(hDC, hBrush);
FloodFill(hDC, 30, 30, RGB(0,0,255));
// Tạo mẫu tô mới để tô hình thứ hai
hBrush = (HBRUSH)CreateHatchBrush(HS_DIAGCROSS, RGB(0, 255, 255));
hOldBrush = (HBRUSH)SelectObject(hDC, hBrush);
FloodFill(hDC, 150, 150, RGB(0, 0, 0));
SelectObject(hDC, hOldBrush);
//Xóa mẫu tô và giải phóng hDC
DeleteObject(hBrush);
```

```
ReleaseDC(hWnd, hDC);
```

3. Nhóm hàm bitmap

a. Nạp ảnh

Có hai cách nạp ảnh bitmap:

❖ Nạp từ bitmap resource trong chương trình:

```
HBITMAP LoadBitmap(HINSTANCE hInstance, LPCTSTR lpBitmapName);
```

Đối số đầu tiên `hInstance` là handle thể hiện của module chứa bitmap resource, có thể là `NULL` nếu bitmap muốn nạp là một bitmap hệ thống. Đó là các bitmap nhỏ sử dụng cho các thành phần giao diện chương trình như close box và các dấu check, các định danh bắt đầu với tiếp đầu ngữ **OBM**. Đối số thứ hai có thể sử dụng macro `MAKEINTRESOURCE` nếu bitmap kết hợp với một định danh kiểu nguyên, hoặc là chuỗi tên bitmap resource. Sau khi sử dụng, dùng hàm **DeleteObject** để giải phóng handle đang tham chiếu đến bitmap đó.

Ví dụ: Nạp bitmap resource, có sử dụng **MDC** để tránh gây chớp màn hình.

```
// Lấy về kích thước vùng client của cửa sổ
RECT rect;
GetClientRect(hWnd, &rect);
// Lấy về handle device context của cửa sổ
HDC hdc;
hdc = GetDC(hWnd);
//Tạo MDC tương thích với hDC của cửa sổ
HDC hMemDC;
hMemDC = CreateCompatibleDC(hdc);
HBITMAP hNewBmp,hOldBmp;
// Nạp bitmap resource có ID là IDB_CUB
hNewBmp = LoadBitmap(hInst, MAKEINTRESOURCE(IDB_CUB));
if (hNewBmp) {
   //Gán bitmap mới cho MDC
   hOldBmp = (HBITMAP)SelectObject(hMemDC,hNewBmp);
   //Copy bitmap từ MDC sang DC
   BitBlt(hdc, 0, 0, rect.right, rect.bottom, hMemDC, 0, 0, SRCCOPY);
}
//Phục hồi lại bitmap cũ cho MDC
SelectObject(hMemDC, hOldBmp);
//Giải phóng bitmap vừa tạo và MDC, hdc
DeleteObject(hNewBmp);
DeleteDC(hMemDC);
ReleaseDC(hdc);
```

❖ Nạp từ file bitmap của Windows(***.bmp**):

- **Bảng màu(Color Palette):**

Bảng màu trong Windows GDI là một khái niệm giống như bảng màu thật sự của họa sĩ, nó gồm các ô màu(*palette entry*) với thông tin cụ thể về một sắc độ.

Để quản lý việc thể hiện màu trên các cửa sổ, Windows sử dụng các bảng màu. Thông thường các cửa sổ dùng bảng màu hệ thống để thể hiện thông tin. Tuy nhiên, trong nhiều trường hợp bảng màu hệ thống không phù hợp với nhu cầu, khi đó ta cần thay một số màu trong bảng màu hệ thống bằng các màu phù hợp hơn.

Trong thực tế, Windows cho phép mỗi cửa sổ dùng một bảng màu khác nhau với điều kiện số lượng màu trong bảng màu đó không được vượt quá số màu tối đa cho phép của thiết bị hiển thị. Khi một cửa sổ nhận sự quan tâm (*focus*), Windows sẽ thể hiện dữ liệu trong cửa sổ đó với số màu tối đa cho phép của bảng màu tương ứng, những màu còn lại trong **palette** sẽ được thể hiện bằng những màu gần giống nhất. Đối với những cửa sổ không active, dữ liệu sẽ được thể hiện bằng những màu gần giống nhất trong bảng màu hệ thống.

Ví dụ về cách thể hiện màu sử dụng palette:

  - Bảng màu hệ thống có 12 giá trị màu.
  - Bảng màu 1 dùng cho cửa sổ 1. Cửa sổ 1 đang active nên các màu của bảng màu 1 được ánh xạ trực tiếp vào bảng màu hệ thống.
  - Bảng màu 2 dùng cho cửa sổ 2. Cửa sổ 2 không active nên các màu 0,2,4,5,6 được ánh xạ vào các vị trí còn trống trên bảng màu hệ thống; các màu còn lại được ánh xạ vào những màu gần giống nhất của bảng màu hệ thống.

- **Cấu trúc file bitmap:**

File bitmap gồm 2 phần: phần header và phần dữ liệu ảnh.

- Cấu trúc **BITMAPFILEHEADER**

Tên trường	Kích thước	Ý nghĩa
bfType	2 bytes(WORD)	Loại file(=**BM**)
bfSize	4 bytes(DWORD)	Kích thước file(bytes)
bfReserved1	2 bytes(WORD)	Không dùng(=0)
bfReserved2	2 bytes(WORD)	Không dùng(=0)
bfOffBits	4 bytes(DWORD)	Kích thước của phần **BitmapHeader** (byte)

- Cấu trúc **BITMAPINFO**

Tên trường	Kiểu	Ý nghĩa
bmiHeader	**BITMAPINFOHEADER**	Các thông số ảnh bitmap
bmiColors	**RGBQUAD***	Bảng màu của ảnh.

- Cấu trúc **BITMAPINFOHEADER**

Tên trường	Kiểu	Ý nghĩa
biSize	DWORD	Kích thước phần **BITMAPINFO**(=40 bytes)
biWidth	LONG	Chiều rộng bitmap(theo đơn vị pixel)
biHeight	LONG	Chiều cao bitmap(theo đơn vị pixel)
biPlanes	WORD	Số plane. Luôn = 1.
biBitCount	WORD	Số bit dùng để lưu giá trị 1 điểm ảnh. Giá trị này cho biết số màu có trong bảng màu bmiColors.
biCompression	DWORD	Hình thức nén của dữ liệu ảnh: – BI_RGB: ảnh không nén – BI_RLE8: nén theo hình thức RLE cho ảnh 8 bits/pixel – BI_RLE4: nén theo hình thức RLE cho ảnh 4 bits/pixel
biSizeImage	DWORD	Kích thước của ảnh(đơn vị byte). Trường hợp biCompression = **BI_RGB** thì giá trị này có thể = 0.
biXPelsPerMeter	LONG	Độ phân giải (**DPI/m**) theo phương ngang. Dùng trong trường hợp ảnh quét từ scanner.
biYPelsPerMeter	LONG	Độ phân giải (**DPI/m**) theo phương dọc. Dùng trong trường hợp ảnh quét từ scanner.
biClrUsed	DWORD	Số màu thực sự được dùng. Nếu = 0 thì có nghĩa là dùng tất cả màu trong bảng màu.
biClrImportant	DWORD	Số lượng màu quan trọng dùng trong việc hiển thị ảnh. Nếu = 0 thì dùng tất cả màu trong bảng màu.

- Cấu trúc **RGBQUAD**

bmiColors là một mảng kiểu **RGBQUAD**, chứa các giá trị màu dùng cho ảnh, đây chính là bảng màu của ảnh. Số lượng phần tử của bmiColors bằng số màu của ảnh. bmiColors chỉ dùng cho ảnh đen trắng, 16 màu và 256 màu; khi đó phần dữ liệu ảnh cho mỗi pixel chỉ là các index đến 1 vị trí trong bảng màu này. Đối với ảnh 16 bits và 24 bits màu, giá trị bmiColors = NULL, bảng màu không được sử dụng nữa. Trong trường hợp này, nếu ảnh là 16 bits thì mỗi pixel sẽ được biểu diễn bằng 1 **WORD** trong phần dữ liệu ảnh, 5 bits thấp nhất sẽ lưu giá trị màu **Blue**, 5 bits kế sẽ lưu giá trị màu **Green**, và 5 bits sau cùng lưu giá trị màu **Red**, bit cao nhất không được sử dụng (Lý do là ảnh dùng 16 bits để biểu diễn **RGB**, ta chia đều cho 3 sắc độ nên mỗi sắc độ **R**, **G**, **B** sẽ có 5 bits để biểu diễn và dư ra 1 bit). Ảnh 24 bits thì dùng 3 bytes để biểu diễn cho 3 sắc độ **RGB** cho mỗi pixel trong phần dữ liệu ảnh.

Tên trường	Kiểu	Ý nghĩa
rgbRed	BYTE	Giá trị thành phần RED
rgbGreen	BYTE	Giá trị thành phần GREEN
rgbBlue	BYTE	Giá trị thành phần BLUE
rgbReserved	BYTE	Không dùng

- Cấu trúc phần dữ liệu ảnh (**BITMAPDATA**):

Như đã trình bày ở trên, phần dữ liệu tập tin bitmap dùng để lưu giá trị index của màu tương ứng với điểm ảnh trong bảng màu đối với ảnh có số màu ≤ 256. Các điểm ảnh được lưu tuần tự từ trái sang phải theo từng dòng quét(*scanline*) theo thứ tự từ dưới lên.

Đối với ảnh <=256 màu, để biết giá trị màu (**R,G,B**) ứng với 1 điểm ảnh (**x,y**), sử dụng công thức sau:

(**R,G,B**) của điểm ảnh i = bmiColors[i]

Đối với ảnh 16 và 24 bits màu, ta không dùng palette và cách biểu diễn của phần dữ liệu đã được trình bày ở trên.

Phần dữ liệu của file bitmap có thể được nén theo phương pháp RLE, tài liệu này không trình bày chi tiết vào mọi loại ảnh nén, nếu cần các bạn có thể tham khảo trong các giáo trình về xử lý ảnh.

- **Cách tạo bitmap từ file *.bmp:**
  - Đọc header của file, lấy các thông tin về ảnh: độ rộng, độ cao, bảng màu…
  - Đọc dữ liệu ảnh và giải mã(nếu cần thiết) vào bộ nhớ đệm.
  - Tạo palette màu từ bảng màu lấy được từ file bằng hàm **CreatePalette**.
  - Đổi palette màu cho thiết bị xuất, sử dụng các hàm **SelectPalette**, **RealizePalette**.
  - Tạo lập bitmap handle bằng hàm **CreateDIBitmap**.

Ví dụ: Nạp ảnh bitmap từ file.

```
void LoadBmp(char *bmpFN, HDC hdc, RECT rect) {
    try {
        int FHandle;
        BITMAPFILEHEADER bmpfh;
        HANDLE hBmp, hBuff;
        BITMAPINFO *BmpInfo;
        unsigned char *Buff;
        int Count,NumColor;
        long n, BytesPerLine;
        Count = 0;
        SetCursor(LoadCursor(NULL, IDC_WAIT));
        // Mở file bitmap để đọc
        if ((Fhandle = open(bmpFN, O_BINARY|O_RDONLY)) == -1)
            throw "Cannot open the file.";

        // Đọc header file bitmap
        read(FHandle, &bmpfh, sizeof(BITMAPFILEHEADER));//Đây chỉ là file header
        hBmp = GlobalAlloc(LMEM_MOVEABLE,
                  bmpfh.bfOffBits - sizeof(BITMAPFILEHEADER));//Kích thước BITMAPHEADER
        BmpInfo = (BITMAPINFO*)GlobalLock(hBmp);
        // Đọc phần bitmap info về palette màu
        read(FHandle, BmpInfo, bmpfh.bfOffBits-sizeof(BITMAPFILEHEADER));Lấy BM INFO
        if (BmpInfo->bmiHeader.biCompression != BI_RGB)
            throw "Program don't support compressed bitmap file.";
        // Xác định số màu của palette
        NumColor = (int)pow(2,BmpInfo->bmiHeader.biBitCount);
        if (NumColor > 256)
            throw "The program supports 256 colors bitmap file only.";
        // Xác định số byte cho 1 scanline
        BytesPerLine = (bmpfh.bfSize-bmpfh.bfOffBits)/BmpInfo->bmiHeader.biHeight;
        /* Cập nhật thông tin kích thước ảnh trên BITMAPINFO
           (hiện tại = 0 nếu ảnh không nén) */
        BmpInfo->bmiHeader.biSizeImage=bmpfh.bfSize-bmpfh.bfOffBits;
        // Đọc nội dung file bitmap
        hBuff = GlobalAlloc(GMEM_MOVEABLE, BmpInfo->bmiHeader.biSizeImage);
        Buff = (unsigned char *)GlobalLock(hBuff);
        if (!Buff)
            throw "Cannot read the bitmap data.";
        // Đọc từng dòng trên file
        do {
            n = read(FHandle,Buff,BytesPerLine);
            Buff+=n;
            Count++;
        }while (Count<BmpInfo->bmiHeader.biHeight);
```

```
        close(FHandle);
        GlobalUnlock(hBuff);
        GlobalUnlock(hBmp);
        SetCursor(LoadCursor(NULL, IDC_ARROW));

        /* Tạo palette m u cho ảnh Bitmap */
        HANDLE hMemPal;
        LOGPALETTE far *logPal;
        HPALETTE hPal;
        int i;
        hMemPal = GlobalAlloc(LMEM_MOVEABLE,
                    sizeof(LOGPALETTE) + NumColor * sizeof(PALETTEENTRY));
        logPal = (LOGPALETTE far *)GlobalLock(hMemPal);
        // S' mục trong Palette
        logPal->palNumEntries = NumColor;
        logPal->palVersion = 0x300;
        //Gán các giá trị m u RGB cho những Entry trong Palette
        for (i = 0; i < NumColor; i++) //Error here
        {
            logPal->palPalEntry[i].peRed = BmpInfo->bmiColors[i].rgbRed;
            logPal->palPalEntry[i].peGreen = BmpInfo->bmiColors[i].rgbGreen;
            logPal->palPalEntry[i].peBlue = BmpInfo->bmiColors[i].rgbBlue;
            logPal->palPalEntry[i].peFlags = PC_RESERVED;
        }
        // Tạo handle cho Palette
        hPal = CreatePalette(logPal);
        GlobalUnlock(hMemPal);
        GlobalFree(hMemPal);
        if (!hPal)
            throw "Cannot create palette.";

        /* Tao handle bmp file */
        SetCursor(LoadCursor(NULL, IDC_WAIT));
        // Map palette v o hdc
        SelectPalette(hdc,hPal,FALSE);
        RealizePalette(hdc);
        BmpInfo = (BITMAPINFO *)LocalLock(hBmp);
        Buff = (unsigned char *)GlobalLock(hBuff);
        //Tạo bitmap
        hBmp = CreateDIBitmap(hdc,(BITMAPINFOHEADER *)BmpInfo, CBM_INIT,
                  Buff, (BITMAPINFO *)BmpInfo,DIB_RGB_COLORS);
        GlobalUnlock(hBuff);
        GlobalUnlock(hBmp);
        GlobalFree(hBuff);
        GlobalFree(hBmp);
        SetCursor(LoadCursor(NULL, IDC_ARROW));
        if (!hBmp)
            throw "Cannot create bitmap.";
        /* Hiển thị bitmap lên cửa s• */
        HDC hMemDC;
        HANDLE OldBmp;
        // Tạo MDC
        hMemDC = CreateCompatibleDC(hdc);
        SelectPalette(hMemDC, hPal, FALSE);
        // Trả về palette hệ th'ng
        UnrealizeObject(hPal);
        // Map lại palette
        RealizePalette(hdc);
        SelectPalette(hMemDC, hPal, FALSE);
```

```
        // Liên kết hMemDC với ảnh bitmap
        OldBmp = SelectObject(hMemDC, hBmp);
        // Copy bitmap lên window
        BitBlt(hdc, 0, 0, rect.right, rect.bottom, hMemDC, 0, 0,SRCCOPY);
        // Hủy MDC
        SelectObject(hMemDC,OldBmp);
        DeleteDC(hMemDC);
    } catch(char *e)
    MessageBox(NULL, e, "Error", MB_OK|MB_ICONERROR);
}
```

b. Zoom ảnh

Thao tác zoom ảnh đòi hỏi phải có hai MDC, một MDC nguồn để copy ảnh vào, một MDC đích để lưu ảnh đã được zoom từ MDC nguồn.

Ví dụ: zoom ảnh bitmap lấy từ bitmap resource.

```
// Lấy về kích thước vùng client của cửa s•
RECT rect;
GetClientRect(hWnd, &rect);
//Lấy về handle device context của cửa s•
HDC hdc;
hdc = GetDC(hWnd);
// Tạo 2 MDC tương thích với DC của cửa s•
HDC hSourceMemDC, hDestMemDC;
hSourceMemDC = CreateCompatibleDC(hdc);
hDestMemDC = CreateCompatibleDC(hdc);
//Nạp bitmap cần load va tạo một bitmap rộng tương thích hdc
HBITMAP hBmp,hOldBmp,hBlankBmp,hOldBlankBmp;
hBmp = LoadBitmap(hInst, MAKEINTRESOURCE(IDB_CUB));
hBlankBmp = CreateCompatibleBitmap(hdc,rect.right,rect.bottom);
if (hBmp && hBlankBmp && hSourceMemDC && hDestMemDC) {
    // Chọn bitmap cần load cho MDC nguồn
    hOldBmp = (HBITMAP)SelectObject(hSourceMemDC, hBmp);
    // Chọn blank bitmap cho MDC 'ích
    hOldBlankBmp =(HBITMAP)SelectObject(hDestMemDC,hBlankBmp);
    // Lấy về thông tin của ảnh
    BITMAP bmpInfo;
    GetObject(hBmp, sizeof(BITMAP),(BITMAP *)&bmpInfo);
    // Zoom ảnh từ MDC nguồn sang MDC 'ích
    StretchBlt(hDestMemDC, 0, 0, rect.right, rect.bottom,hSourceMemDC, 0, 0,
bmpInfo.bmWidth, bmpInfo.bmHeight, SRCCOPY);
    // Copy ảnh 'ã zoom từ MDC 'ích lên hdc
    BitBlt(hdc, 0, 0, rect.right, rect.bottom, hDestMemDC, 0,0,SRCCOPY);
}
// Giải phóng các bitmap handle, MDC handle
SelectObject(hSourceMemDC, hOldBmp);
DeleteObject(hBmp);
SelectObject(hDestMemDC, hOldBlankBmp);
DeleteObject(hBlankBmp);
DeleteDC(hSourceMemDC);
DeleteDC(hDestMemDC);
ReleaseDC(hdc);
```

D. IN ẤN

Trên Windows, các ứng dụng không truy xuất trực tiếp máy in mà sẽ dùng các hàm GDI để vẽ ra device context của máy in. Do đó việc in ấn trên Windows trở nên đơn giản, tương tự việc vẽ ra các hộp thoại trên màn hình.

Với cơ chế này, người lập trình không cần quan tâm đến việc máy in đang sử dụng là máy in hiệu gì, loại nào… Các Windows **printer device driver** sẽ đảm nhận việc chuyển dữ liệu cần in sang dạng phù hợp với máy in hiện hành.

Tiến trình in ấn trên Windows bao gồm các bước sau:

- Lấy về device context của máy in cần in.
- Gởi yêu cầu bắt đầu việc in ấn (**STARTDOC**).
- Kết xuất dữ liệu ra máy in(có thể sang trang _ **NEWFRAME** nếu cần).
- Gởi yêu cầu kết thúc việc in ấn(**ENDDOC**).
- Giải phóng device context máy in.

1. Lấy về handle device context máy in

a. Sử dụng file win.ini

Dựa vào thông tin về máy in mặc định trong file **win.ini**. File win.ini lưu thông tin về máy in trong section [windows], entry DEVICE.

Ví dụ: một đoạn của file win.ini chứa thông tin về máy in mặc định.

```
[windows]
device = EpsonFX1050, Epson9, LPT1
```

Đó là các thông tin về tên máy in, driver, và cổng giao tiếp với máy tính.

Một số hàm chuẩn để đọc ghi file win.ini của Windows:

```
DWORD GetProfileString(LPCTSTR lpAppName, LPCTSTR lpKeyName, LPCTSTR lpDefault,
                       LPTSTR lpReturnedString, DWORD nSize);
BOOL WriteProfileString(LPCTSTR lpAppName, LPCTSTR lpKeyName, LPCTSTR lpString);
```

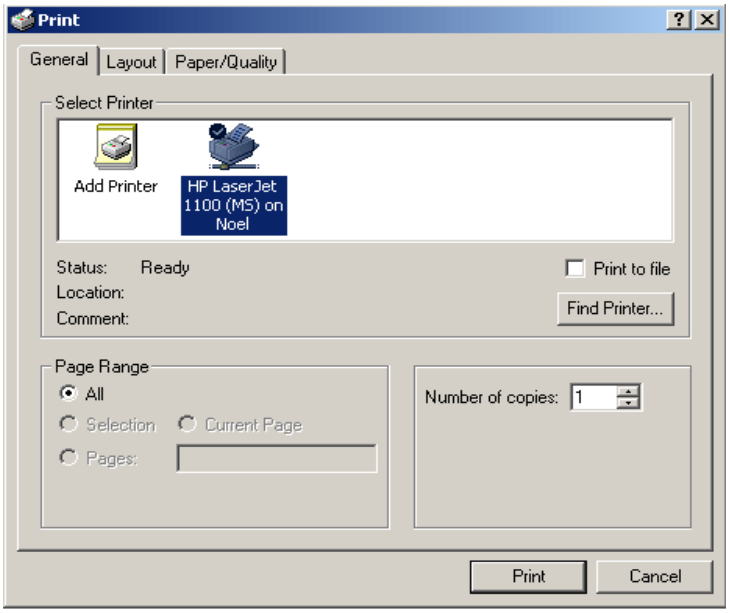
Ví dụ: In dòng chữ ra máy in mặc định.

```
void PrintStrWinInit(char *msg) {
    char prnInfo[200];
    GetProfileString("windows", "device", "NoPrinter", prnInfo, 200);
    if (strcmp(prnInfo, "NoPrinter")) {
        char prnName[100], prnDriver[100], prnPort[100];
        char seps[10];
        char *token;
        strcpy(seps,",");
        token = strtok(prnInfo, seps);
        strcpy(prnName, token);
        token = strtok(NULL, seps);
        strcpy(prnDriver, token);
        token = strtok(NULL, seps);
        strcpy(prnPort, token);
        HDC prnDC;
        prnDC = CreateDC(prnDriver, prnName, prnPort, NULL);
        if (prnDC) {
            StartDoc(prnDC, NULL);
            StartPage(prnDC);
            TextOut(prnDC, 100, 100, "Your message here:", 18);
            TextOut(prnDC, 100, 200, msg, strlen(msg));
            TextOut(prnDC, 100, 300, "End of printing", 15);
            EndPage(prnDC);
            EndDoc(prnDC);
            DeleteDC(prnDC);
        }
    }
}
```

b. Sử dụng hộp thoại Print

Một cách khác đơn giản hơn để lấy về handle device context của một máy in bất kỳ thay vì máy in mặc định là sử dụng hộp thoại in chuẩn của Windows.

Hộp thoại in chuẩn được kích hoạt bằng hàm **PrintDlg** (hàm này được định nghĩa trong module COMMDLG.DLL), hộp thoại này cho phép người dùng thiết lập các thông số in ấn.



Hình 6.4 Hộp thoại in

Kết quả trả về của hàm **PrintDlg** là một handle device context của máy in đã chọn.

Ví dụ: In dòng chữ ra máy in được chọn từ hộp thoại in chuẩn.

```
void PrintStrPrintDlg(char *msg) {
    PRINTDLG pd;
    ZeroMemory(&pd, sizeof(PRINTDLG));
    pd.lStructSize = sizeof(PRINTDLG);
    pd.hwndOwner = GetFocus();
    pd.Flags = PD_RETURNDC|PD_NOSELECTION;
    if (PrintDlg(&pd)) {
        StartDoc(pd.hDC, NULL);
        StartPage(pd.hDC);
        TextOut(pd.hDC, 100, 100, "Your message here:", 18);
        TextOut(pd.hDC, 100, 200, msg, strlen(msg));
        TextOut(pd.hDC, 100, 300, "End of printing", 15);
        EndPage(pd.hDC);
        EndDoc(pd.hDC);
        DeleteDC(pd.hDC);
    }
}
```

2. Các hàm điều khiển in ấn

Hàm	Ý nghĩa
int StartDoc(hDC, lpDocInfo)	Bắt đầu tác vụ in ấn 1 tài liệu trong **file** có tên lpDocInfo, hDC là handle printer device context.
int EndDoc(hDC)	Kết thúc tác vụ in ấn.
int AbortDoc(hDC)	Hủy tác vụ in hiện hành.
int StartPage(hDC)	Bắt đầu trang mới.
int EndPage(hDC)	Kết thúc 1 trang.
int SetAbortDoc(hDC, MyAProc)	Chỉ định hàm (MyAProc) xử lý trường hợp người dùng muốn ngắt ngang tác vụ in.

Bảng 6.2 Các hàm điều khiển in ấn

3. Xử lý lỗi khi in

Trong quá trình in bằng việc sử dụng các hàm điều khiển in ấn, nếu các thao tác thành công thì luôn trả về giá trị lớn hơn 0.

Nếu có lỗi, dùng hàm **DWORD GetLastError**(**VOID**) để lấy về mã lỗi (chỉ WinNT và Win2000 mới hỗ trợ hàm này).

CHƯƠNG VII: QUẢN LÝ BỘ NHỚ VÀ TẬP TIN

A. MỞ ĐẦU

Một tiến trình thông thường được hiểu là một chương trình máy tính đang được thi hành. Quá trình thực thi các tiến trình gắn liền với việc quản lý và sử dụng các tài nguyên trong máy tính, trong đó đáng kể nhất là quản lý bộ nhớ chính và xử lý hệ thống lưu trữ phụ.

Trong các hệ thống máy tính hiện đại, bộ nhớ chính là trung tâm của các thao tác, xử lý. Bộ nhớ chính có thể xem như một mảng các phần tử kiểu BYTE hoặc WORD, được xác định thông qua địa chỉ của chúng. Mỗi chương trình được ánh xạ vào bộ nhớ chính trước khi được thi hành và được hệ điều hành quản lý thông qua tập lệnh xác định.

Trong suốt quá trình thi hành, các chương trình với dữ liệu truy xuất của chúng luôn được đặt trong bộ nhớ chính. Nhưng bộ nhớ chính thì khá nhỏ để có thể lưu giữ mọi dữ liệu và chương trình, ngoài ra dữ liệu sẽ mất khi không còn được cung cấp năng lượng. Do đó, cần phải sử dụng hệ thống lưu trữ phụ.

Chương này trình bày các vấn đề trên qua hai phần sau: phần B - *Quản lý bộ nhớ* - trình bày cách thức Microsoft® Win32® API quản lý các vùng nhớ thông qua các hàm cấp phát, sử dụng, và giải phóng chúng; cách thức thao tác trên địa chỉ vùng nhớ ảo và các trang nhớ. Phần C - *Xử lý tập tin* - trình bày các hàm thực hiện các thao tác tạo, xử lý và hủy tập tin, cũng như tìm hiểu một số vấn đề liên quan đến tập tin.

B. QUẢN LÝ BỘ NHỚ

Mỗi tiến trình trong Win32 đều có một vùng địa chỉ ảo 32-bit cho phép định vị vùng nhớ đến 4 GB. Địa chỉ ảo này không phải là vùng nhớ vật lý thực tế. Windows sử dụng một cấu trúc dữ liệu ánh xạ để chuyển đổi địa chỉ ảo thành vùng nhớ vật lý.

Vùng địa chỉ ảo của mỗi tiến trình thường lớn hơn rất nhiều so với vùng nhớ vật lý thực sự trên máy tính. Do đó, để tăng vùng nhớ cho các tiến trình đang thực hiện, hệ thống sử dụng vùng nhớ trống trên đĩa. Vùng nhớ vật lý và vùng địa chỉ ảo của mỗi tiến trình được tổ chức thành các trang, phụ thuộc vào họ máy tính. Ví dụ, đối với máy tính họ x86, mỗi trang có kích thước là 4 KB.

Để tăng khả năng linh động trong việc quản lý bộ nhớ, hệ thống có thể di chuyển các trang từ bộ nhớ chính vào đĩa và ngược lại. Các thao tác này được thực hiện chỉ bởi hệ thống, các ứng dụng chỉ việc gọi các hàm cấp phát và sử dụng vùng địa chỉ ảo.

Thư viện C chuẩn hỗ trợ các hàm cấp phát và giải phóng vùng nhớ như **malloc**, **free**, …, hoặc trong C++ là **new**, **delete**, …. Thế nhưng trong Windows 16 bits, các hàm này có thể gây lỗi hệ thống. Trong Win32, ta có thể sử dụng chúng an toàn do hệ thống chỉ quản lý bộ nhớ qua các trang vật lý mà không ảnh hưởng đến địa chỉ ảo. Hơn nữa, Win32 cũng không phân biệt giữa con trỏ gần và con trỏ xa. Mặc dù vậy, các hàm trên không thể hiện đủ các khả năng hỗ trợ của việc quản lý bộ nhớ trong Win32. Chúng ta sẽ làm quen với các hàm **Global** và **Local** - sử dụng từ Windows 16 bits, và các hàm quản lý vùng nhớ ảo khác.

1. Các hàm Global và Local

Các hàm toàn cục (*global*) và địa phương (*local*) là các hàm heap Windows 16 bits. Tuy nhiên, quản lý bộ nhớ trong Win32 cũng hỗ trợ các hàm này để có thể sử dụng các chương trình, hoặc source code của các chương trình viết cho Windows 16 bits. Các hàm toàn cục và địa phương xử lý chậm và ít chức năng hơn các hàm quản lý bộ nhớ mới thiết kế cho Win32. Chúng ta sẽ làm quen các hàm mới ở phần sau.

Để cấp phát vùng nhớ cho một tiến trình, ta có thể sử dụng hàm **GlobalAlloc** hoặc **LocalAlloc**. Việc quản lý vùng nhớ trong Win32 không phân biệt hàm toàn cục hay cục bộ như trong Windows 16 bits. Do đó, không có sự phân biệt giữa các đối tượng vùng nhớ được cấp phát bởi hai hàm trên. Thêm vào đó, việc chuyển mô hình đoạn vùng nhớ 16 bits sang vùng địa chỉ ảo 32 bits thực hiện một số hàm toàn cục và địa phương với các chọn lựa (*options*) không cần thiết hoặc vô nghĩa. Ví dụ, vì cả cấp phát toàn cục và địa phương đều trả về địa chỉ ảo 32 bits, do đó không xác định dạng con trỏ gần hoặc xa trong các hàm trên.

Hai hàm này cấp phát một vùng nhớ theo kích thước `nBytes` trong heap. Có prototype như sau:

```
HGLOBAL GlobalAlloc(UINT uFlags, DWORD nBytes);
HLOCAL LocalAlloc(UINT uFlags, UINT nBytes);
```

Trong đó `uFlags` xác định cách thức cấp phát vùng nhớ. Ta có bảng sau:

Toàn cục	Địa phương	Ý nghĩa
GMEM_FIXED	LMEM_FIXED	Cấp phát vùng nhớ cố định. Giá trị trả về là một con trỏ.
GMEM_MOVEABLE	LMEM_MOVEABLE	Cấp phát vùng nhớ không cố định. Trong Win32, khối nhớ không bao giờ di chuyển trong vùng nhớ vật lý, nhưng trong heap mặc định. Hàm trả về handle của một đối tượng bộ nhớ. Ta dùng hàm **GlobalLock** hoặc **LocalLock** để chuyển handle sang con trỏ vùng nhớ.

GMEM_ZEROINIT	LMEM_ZEROINIT	Khởi tạo nội dung vùng nhớ với giá trị 0.
GPTR		GMEM_FIXED | GMEM_ZEROINIT
GHND		GMEM_MOVEABLE | GMEM_ZEROINIT
	LPTR	LMEM_FIXED | LMEM_ZEROINIT
	LHND	LMEM_MOVEABLE | LMEM_ZEROINIT

Bảng 7.1 Các cờ sử dụng trong các hàm GlobalAlloc và LocalAlloc

Chú ý: Không thể sử dụng giá trị GMEM_FIXED đồng thời với GMEM_MOVEABLE, hoặc LMEM_FIXED đồng thời với LMEM_MOVEABLE.

Nếu thành công, hàm trả về handle cho đối tượng vùng nhớ được cấp phát. Ngược lại, giá trị trả về là NULL.

Các đối tượng vùng nhớ được cấp phát bằng hàm **GlobalAlloc** và **LocalAlloc** là các trang riêng, truy cập đọc-ghi bởi chính tiến trình tạo nó. Các tiến trình khác không thể truy cập các đối tượng vùng nhớ này.

Khi dùng cách thức cấp phát GMEM_MOVEABLE hoặc LMEM_MOVEABLE, ta nhận được handle vùng nhớ. Để sử dụng vùng nhớ, ta dùng hàm **GlobalLock** hoặc **LocalLock:**

```
LPVOID GlobalLock(HGLOBAL hMem);
LPVOID LocalLock(HLOCAL hMem);
```

Nếu thành công hàm trả về con trỏ trỏ đến byte đầu tiên trong khối nhớ. Ngược lại, giá trị trả về là NULL.

Khi khoá (*lock*) vùng nhớ, các khối nhớ không thể dịch chuyển trong bộ nhớ máy tính. Sau khi sử dụng con trỏ vùng nhớ, cần mở khoá (*unlock*) chúng, để hệ thống có thể di chuyển và sử dụng các vùng nhớ linh động cho các tiến trình khác. Ta dùng hai hàm tương ứng là **GlobalUnlock** và **LocalUnlock**.

```
BOOL GlobalUnlock(HGLOBAL hMem);
BOOL LocalUnlock(HLOCAL hMem);
```

Mỗi lần khoá vùng nhớ, biến đếm tương ứng tăng một đơn vị. Mỗi lần mở khoá, biến đếm giảm một. Nếu vùng nhớ còn khoá, hàm trả về giá trị khác 0, ngược lại giá trị trả về là 0.

Kích thước thật sự của vùng nhớ được cấp phát có thể lớn hơn kích thước yêu cầu (nBytes). Để xác định số byte thật sự được cấp phát, ta dùng hàm **GlobalSize** và **LocalSize**.

```
DWORD GlobalSize(HGLOBAL hMem);
UINT LocalSize(HLOCAL hMem);
```

Nếu thành công, hàm trả về số byte kích thước vùng nhớ xác định bởi hMem. Ngược lại, giá trị trả về là 0.

Ngoài ra, ta có thể sử dụng hàm **GlobalReAlloc** và **LocalReAlloc** để cấp phát thay đổi kích thước hoặc thuộc tính vùng nhớ.

```
HGLOBAL GlobalReAlloc(HGLOBAL hMem, DWORD nBytes, UINT uFlags);
HLOCAL LocalReAlloc(HLOCAL hMem, UINT nBytes, UINT nFlags);
```

Trường nBytes xác định kích thước cấp phát lại cho vùng nhớ hMem. Tuy nhiên, khi nFlags chứa GMEM_MODIFY (hoặc LMEM_MODIFY), hệ thống bỏ qua giá trị này. Khi đó, hàm thay đổi các thuộc tính của vùng nhớ.

Để xác định handle của vùng nhớ khi biết con trỏ vùng nhớ, ta dùng hàm **GlobalHandle** và **LocalHandle** như sau:

```
HGLOBAL GlobalHandle(LPCVOID pMem);
HLOCAL LocalHandle(LPCVOID pMem);
```

Với pMem là con trỏ trỏ đến byte đầu tiên trong vùng nhớ. Nếu thành công, hàm trả về handle cần tìm. Ngược lại, giá trị trả về là NULL.

Sau khi sử dụng xong, ta dùng hàm **GlobalFree** và **LocalFree** để giải phóng các vùng nhớ đã được cấp phát.

```
HGLOBAL GlobalFree(HGLOBAL hMem);
HLOCAL LocalFree(HLOCAL hMem);
```

Nếu thành công, giá trị trả về là NULL. Ngược lại, hàm trả về giá trị handle của đối tượng ban đầu.

Đoạn chương trình sau minh họa cách hệ thống cấp phát một vùng nhớ với kích thước yêu cầu là 3500 bytes. Sau đó gán các giá trị vùng nhớ bằng 0x3C.

```
HANDLE hMem;
LPBYTE lpAddress;
int i, nSizeMem;
hMem = GlobalAlloc(GMEM_MOVEABLE, 3500);
```

```
if (hMem != NULL) {
    /* Vùng nhớ có thể lớn hơn 3500 */
    nSizeMem = GlobalSize(hMem);
    lpAddress = (LPBYTE)GlobalLock(hMem);
    if (lpAddress != NULL) {
        for (i = 0; i < nSizeMem; i++)
            lpAddress[i] = 0x3C;
    GlobalUnlock(hMem);
/* … */
}
/* Nếu không dùng nữa thì gọi hàm
GlobalFree(hMem); */
}
```

Đoạn chương trình tiếp theo cấp phát lại vùng nhớ trên với kích thước là 5000 bytes, khởi gán các giá trị là 0x00:

```
HANDLE hMemTmp;
hMemTmp = GlobalReAlloc(hMem, 5000, GMEM_MOVEABLE|GMEM_ZEROINIT);
if (hMemTmp != NULL) {
    hMem = hMemTmp;
    nSizeMem = GlobalSize(hMem);
/* … */
}
…
/* Khi kết thúc sử dụng, cần gọi hàm
GlobalFree(hMem); */
```

2. Các hàm Heap

Các hàm heap cho phép các tiến trình tạo một vùng heap riêng cho một hoặc một số trang trong vùng địa chỉ của tiến trình đang thực hiện. Sau đó tiến trình có thể sử dụng một tập các hàm khác nhau để quản lý vùng nhớ trong heap này. Ở đây không có sự phân biệt giữa vùng nhớ được cấp phát bởi hàm heap riêng hay dùng các hàm cấp phát khác.

Đầu tiên hàm **HeapCreate** tạo đối tượng heap cho một tiến trình. Vùng nhớ heap này chỉ được dùng cho tiến trình này mà thôi, và không chia sẻ cho các tiến trình khác, ngay cả các tiến trình trong thư viện liên kết động DLL (*dynamic-link library*).

```
HANDLE HeapCreate(DWORD flOptions, DWORD dwInitialSize, DWORD dwMaximumSize);
```

Trường flOptions xác định các thuộc tính được chọn cho vùng heap mới được khởi tạo. Có thể là HEAP_GENERATE_EXCEPTIONS và HEAP_NO_SERIALIZE. Trường dwInitialSize xác định kích thước khởi tạo của heap, được làm tròn cho các trang vùng nhớ. Trường dwMaximumSize xác định vùng nhớ tối đa có thể cấp phát cho tiến trình bằng hàm **HeapAlloc** hoặc **HeapReAlloc**. Hàm trả về handle của đối tượng heap nếu thành công, ngược lại trả về NULL.

Để cấp phát vùng nhớ lần đầu, ta gọi hàm **HeapAlloc**. Nếu muốn cấp phát lại, dùng hàm **HeapReAlloc**.

```
LPVOID HeapAlloc(HANDLE hHeap, DWORD dwFlags, DWORD dwBytes);
```

Trường dwFlags có thể là HEAP_GENERATE_ EXCEPTIONS, HEAP_NO_SERIALIZE, và HEAP_ZERO_ MEMORY. Trường dwBytes xác định số bytes vùng heap được cấp phát. Nếu thành công, hàm trả về con trỏ đến vùng nhớ. Nếu thất bại, hàm trả về NULL nếu dwFlags không thiết lập HEAP_GENERATE_EXCEPTIONS. Nếu có thiết lập, giá trị trả về là STATUS_NO_MEMORY (không có sẵn vùng nhớ hoặc lỗi vùng heap), hoặc STATUS_ACCESS_VIOLATION (Do lỗi vùng heap hoặc biến không chính xác).

```
LPVOID HeapReAlloc(HANDLE hHeap, DWORD dwFlags, LPVOID lpMem, DWORD dwBytes);
```

Trường lpMem trỏ đến vùng nhớ cần cấp phát lại. Vùng nhớ này đã được tạo bằng hàm **HeapAlloc** hoặc **HeapReAlloc**.

Trường dwBytes xác định kích thước vùng nhớ cần cấp phát. Giá trị này phải nhỏ hơn 0x7FFF8.

Để khoá và mở khoá vùng nhớ heap, ta dùng hàm **HeapLock** và **HeapUnlock**.

```
BOOL HeapLock(HANDLE hHeap);
BOOL HeapUnlock(HANDLE hHeap);
```

Nếu thành công, giá trị trả về khác 0. Ngược lại, hàm trả về 0.

Để xác định kích thước vùng heap, ta dùng hàm HeapSize.

```
DWORD HeapSize(HANDLE hHeap, DWORD dwFlags, LPCVOID lpMem);
```

Trong hàm này, dwFlags chỉ dùng với HEAP_NO_SERIALIZE. Các trường khác tương tự các hàm khác. Nếu thành công, hàm trả về kích thước vùng nhớ. Nếu thất bại, hàm trả về giá trị là 0xFFFFFFFF.

Sau khi sử dụng, ta giải phóng vùng nhớ và hủy đối tượng heap bằng hàm **HeapFree** và **HeapDestroy**.

```
BOOL HeapFree(HANDLE hHeap, DWORD dwFlags, LPVOID lpMem);
BOOL HeapDestroy(HANDLE hHeap);
```

Trong đó, trường dwFlags được định nghĩa chỉ với giá trị HEAP_NO_SERIALIZE. Nếu thành công, hai hàm này đều trả về giá trị khác 0. Ngược lại, giá trị trả về là 0.

Chúng ta không minh họa các hàm sử dụng bộ nhớ heap trong tài liệu này.

3. Các hàm Virtual

Microsoft® Win32® API cung cấp một tập các hàm quản lý bộ nhớ ảo cho phép một tiến trình thao tác và xác định các trang trong vùng địa chỉ không gian ảo, gồm các chức năng sau:

- Để dành vùng không gian địa chỉ ảo cho một tiến trình. Vùng không gian để dành không cấp phát vùng lưu trữ vật lý thật sự, nhưng ngăn không cho các thao tác cấp phát khác sử dụng vùng nhớ này. Nó không ảnh hưởng đến các tiến trình khác. Khi cần sử dụng, tiến trình sẽ cấp phát vùng lưu trữ vật lý cho không gian này.
- Cấp phát xác nhận chuỗi các trang để dành trong không gian địa chỉ ảo của tiến trình để có thể sử dụng vùng lưu trữ vật lý (trong RAM hoặc đĩa).
- Thiết lập các thuộc tính đọc-ghi, chỉ đọc, hoặc không được truy cập cho các trang đã xác nhận. Điều này khác với các hàm cấp phát chuẩn luôn cấp phát cấp phát các trang với thuộc tính là đọc-ghi.
- Giải phóng chuỗi các trang để dành, để sẵn vùng địa chỉ ảo cho các thao tác cấp phát của tiến trình đang gọi.
- Khử xác nhận các trang đã xác nhận bằng cách giải phóng vùng lưu trữ vật lý, để sẵn cho các thao tác cấp phát của các tiến trình khác.
- Khoá một hoặc một vài trang vùng nhớ đã xác nhận vào vùng nhớ vật lý (RAM) để hệ thống có thể hoán chuyển các trang vào tập tin trang.
- Nhận thông tin về chuỗi các trang trong vùng địa chỉ ảo của tiến trình đang gọi hoặc của một tiến trình xác định khác.
- Thay đổi các chức năng bảo vệ truy cập cho chuỗi xác định các trang đạ xác nhận trong vùng địa chỉ ảo của tiến trình đang gọi hoặc tiến trình xác định khác.

a. Cấp phát vùng nhớ ảo

Các hàm quản lý bộ nhớ ảo thực hiện các thao tác trên các trang vùng nhớ. Để cấp phát các trang vùng nhớ ảo, ta dùng hàm **VirtualAlloc**, với các chức năng sau đây:

- Để dành một hay nhiều trang trống.
- Cấp phát xác nhận một hay nhiều trang để dành.
- Để dành và cấp phát xác nhận một hay nhiều trang trống.

Chúng ta có thể chỉ định địa chỉ đầu của các trang để dành hay cấp phát, hoặc để cho hệ thống tự xác nhận địa chỉ. Hàm sẽ làm tròn địa chỉ chỉ định với biên trang thích hợp. Vùng nhớ được cấp phát được khởi gán bằng 0, nếu ta không thiết lập cờ MEM_RESET.

```
LPVOID VirtualAlloc(LPVOID lpAddress, DWORD dwSize,
                    DWORD flAllocationType, DWORD flProtect);
```

Trường lpAddress xác định địa chỉ bắt đầu của vùng cấp phát. Nếu vùng nhớ đang để dành, địa chỉ chỉ định được làm tròn đến biên 64 KB kế tiếp. Nếu vùng nhớ đã để dành và đang được xác nhận, địa chỉ sẽ được làm tròn đến biên trang kế. Để xác định kích thước của trang, ta sử dụng hàm **GetSystemInfo**. Nếu biến này bằng NULL, hệ thống tự xác nhận địa chỉ vùng nhớ cấp phát.

Trường dwSize xác định số byte kích thước vùng nhớ. Nếu lpAddress bằng NULL, giá trị này sẽ được làm tròn đến biên trang kế. Nếu không, các trang cấp phát là các trang chứa một hay nhiều byte nằm trong khoảng từ lpAddress đến lpAddress+dwSize. Nghĩa là, nếu hai byte nằm ở hai trang thì cả hai trang đó đều nằm trong vùng cấp phát.

Trường flAllocationType xác định dạng cấp phát, có thể kết hợp từ các cờ:

Cờ	Ý nghĩa
MEM_COMMIT	Cấp phát vùng lưu trữ vật lý trong bộ nhớ hoặc đĩa. Các trang đã được cấp phát xác nhận hoặc khử cấp phát đều có thể được cấp phát lại mà không gây ra lỗi.
MEM_RESERVE	Để dành vùng không gian địa chỉ ảo của tiến trình. Không thể cấp phát vùng để dành bằng

	các hàm cấp phát bộ nhớ khác (**malloc**, **GlobalAlloc**, …) cho đến khi chúng được giải phóng. Chúng chỉ được cấp phát bằng hàm **VirtualAlloc**.
MEM_RESET	Áp dụng cho Windows NT. Khi thiết lập với giá trị này, dữ liệu được xem như không quan trọng, có thể bị viết chồng lên. Ứng dụng không hoán chuyển dữ liệu từ bộ nhớ chính vào (ra) tập tin trang. Mặt khác, khi thiết lập giá trị này, hệ thống sẽ bỏ qua các giá trị của flProtect.
MEM_TOPDOWN	Cấp phát vùng nhớ tại địa chỉ cao nhất có thể.

Bảng 7.2 Các cờ xác định dạng cấp phát flAllocationType.

Trường flProtect xác định cách thức bảo vệ truy cập vùng nhớ. Nếu các trang đã được cấp phát xác nhận, một trong các cờ sau có thể được thiết lập, kết hợp với các cờ PAGE_GUARD và PAGE_NOCACHE:

Cờ	Ý nghĩa
PAGE_READONLY	Chỉ cho phép đọc các trang cấp phát (không được ghi).
PAGE_READWRITE	Cho phép truy cập đọc và ghi các trang vùng nhớ.
PAGE_EXECUTE	Cho phép thực thi các tiến trình, nhưng không đọc và ghi.
PAGE_EXECUTE_READ	Cho phép thực thi và đọc, nhưng không được ghi.
PAGE_EXECUTE_READWRITE	Cho phép thực thi, đọc và ghi.
PAGE_GUARD	Các trang trong vùng trở thành các trang "lính canh". Nếu ghi hoặc đọc các trang này, hệ thống sẽ phát sinh lỗi ngoại lệ STATUS_PAGE_GUARD và tắt tình trạng đó của trang "lính canh". Xem thêm ở ví dụ trong phần 7.2.3.4.
PAGE_NOACCESS	Cấm truy cập (đọc, ghi, thực thi) các trang. Nếu truy cập, ta có lỗi bảo vệ chung.
PAGE_NOCACHE	Không dùng bộ nhớ đệm. Thích hợp với các chế độ bảo vệ trang hơn là NO_ACCESS.

Bảng 7.3 Các cờ xác định dạng bảo vệ truy cập flProtect.

Nếu thành công, hàm trả về địa chỉ cơ sở của các trang vùng cấp phát. Ngược lại giá trị trả về là NULL.

b. Giải phóng vùng nhớ ảo

Để giải phóng vùng nhớ ảo, ta dùng hàm **VirtualFree**. Hàm giải phóng hoặc khử cấp phát (hoặc cả hai) các trang trong không gian địa chỉ ảo của tiến trình đang gọi.

```
BOOL VirtualFree(LPVOID lpAddress, DWORD dwSize, DWORD dwType);
```

Trường lpAddress là con trỏ trỏ đến vùng các trang cần giải phóng. Nếu dwType chứa cờ MEM_RELEASE, đây phải là con trỏ trả về từ hàm VirtualAlloc.

Trường dwSize xác định số byte kích vùng nhớ cần giải phóng. Nếu dwType chứa cờ MEM_RELEASE, giá trị này cần thiết lập bằng 0. Trong các trường hợp khác, vùng ảnh hưởng sẽ là các trang có ít nhất một byte nằm trong đoạn lpAddress đến lpAddress + dwSize. Nghĩa là, nếu có 2 byte nằm ở biên hai trang khác nhau, thì cả hai trang đều được giải phóng.

Trường dwType xác định cách giải phóng, sử dụng giá trị MEM_DECOMMIT, hoặc MEM_RELEASE. Với giá trị đầu, hàm giải phóng các trang chỉ định (đã được xác nhận cấp phát). Nếu các trang chưa được cấp phát, ta vẫn có thể khử cấp phát (*decommit*) mà không gây ra lỗi. Với giá trị sau, hàm giải phóng vùng nhớ để dành. Trong trường hợp này, dwSize phải bằng 0, nếu không hàm thực hiện thất bại.

Nếu thành công, hàm trả về giá trị khác 0. Ngược lại, giá trị trả về là 0.

Lưu ý để giải phóng các trang, các trang phải cùng tình trạng (cấp phát hay để dành), và tất cả các trang để dành bằng hàm cấp phát **VirtualAlloc** cần giải phóng đồng thời. Nếu một số trang để dành ban đầu đã được xác nhận cấp phát, chúng cần được khử cấp phát trước khi gọi hàm **VirtualFree** để giải phóng.

c. Thao tác trên các trang vùng nhớ

Để xác định kích thước các trang trên máy tính, ta sử dụng hàm **GetSystemInfo**.

```
VOID GetSystemInfo(LPSYSTEM_INFO lpSystemInfo);
```

Trường lpSystemInfo trỏ đến cấu trúc **SYSTEM_INFO** chứa các thông tin hệ thống.

```
typedef struct _SYSTEM_INFO {
    union {
        DWORD dwOemId;
        struct {
            WORD wProcessorArchitecture;
            WORD wReserved;
        }
    };
```

```
    DWORD dwPageSize;
    LPVOID lpMinimumApplicationAddress;
    LPVOID lpMaximumApplicationAddress;
    DWORD dwActiveProcessorMask;
    DWORD dwNumberOfProcessors;
    DWORD dwProcessorType;
    DWORD dwAllocationGranularity;
    WORD wProcessorLevel;
    WORD wProcessorRevision;
}SYSTEM_INFO;
```

Để xác định thông tin về bộ nhớ, ta chỉ khảo sát một số trường liên quan. Trường `dwPageSize` các định kích thước các trang theo dạng đã được cấp phát bằng hàm **VirtualAlloc**. Trường `lpMinimumApplicationAddress` trỏ đến địa chỉ vùng nhớ thấp nhất, và trường `lpMaximumApplicationAddress` trỏ đến địa chỉ vùng nhớ cao nhất có thể truy cập bởi các ứng dụng và thư viện liên kết động. Trường `dwAllocationGranularity` xác định độ phân nhỏ mà vùng nhớ ảo cấp phát. Cụ thể, hàm **VirtualAlloc** yêu cầu cấp phát một byte sẽ để dành một vùng không gian bộ nhớ có kích thước là `dwAllocationGranularity` bytes.

Tiến trình có thể khoá một hay nhiều trang đã được cấp phát (xác nhận) vào vùng nhớ vật lý (RAM), ngăn chặn việc hệ thống hoán chuyển các trang vào (ra) tập tin trang bằng cách dùng hàm **VirtualLock**.

```
BOOL VirtualLock(LPVOID lpAddress, DWORD dwSize);
```

Để mở khoá các trang đã bị khoá, ta dùng hàm **VirtualUnlock**, cho phép các trang có thể được hoán chuyển vào (ra) tập tin trang trên đĩa.

```
BOOL VirtualUnlock(LPVOID lpAddress, DWORD dwSize);
```

Trường `lpAddress` trỏ đến địa chỉ cơ sở của vùng các trang cần được khoá. Trường `dwSize` xác định số byte vùng nhớ cần khoá, gồm các trang chứa tất cả các địa chỉ từ `lpAddress` đến `lpAddress + dwSize`.

Nếu thành công, giá trị trả về khác 0. Ngược lại, các hàm trả về 0.

Số trang mặc định được cấp phát tối đa là 30 trang. Tuy nhiên, chúng ta có cũng thể thay đổi số trang tối đa này.

Các trang cần mở khoá không nhất thiết phải là các trang của lần gọi khoá bằng hàm **VirtualLock** trước đó, nhưng đều phải là các trang đang bị khoá.

Khác với các hàm **GlobalLock** và **LocalLock** có dùng một biến đếm để đếm chuỗi các lần khoá vùng nhớ, hàm **VirtualLock** thì không. Do đó để mở khóa, ta chỉ cần gọi hàm **VirtualUnlock** một lần mà thôi.

d. Sử dụng các hàm quản lý bộ nhớ ảo

Trong phần này, chúng ta minh họa bằng ví dụ thực hiện thao tác để dành và xác nhận vùng nhớ, và ví dụ tạo trang "lính canh".

Trong ví dụ đầu tiên, ta sử dụng hàm **VirtualAlloc** và **VirtualFree** để cấp phát để dành và xác nhận vùng nhớ ảo. Đầu tiên, hàm **VirtualAlloc** được gọi để cấp phát để dành một khối các trang. Ta sử dụng giá trị `NULL` cho địa chỉ cơ sở, đồng nghĩa với việc để cho hệ thống tự xác định vị trí vùng cấp phát. Sau đó sử dụng lại hàm **VirtualAlloc** để cấp phát xác nhận các trang trong vùng để dành. Khi đó, ta cần chỉ định địa chỉ cơ sở cho các trang này.

Trong ví dụ này, ta sử dụng cấu trúc **try-except** để xác nhận các trang trong vùng để dành. Mỗi khi có lỗi trang xuất hiện trong quá trình thực hiện khối try, hàm lọc trước khối **except** sẽ được thực hiện. Nếu hàm lọc có thể cấp phát một trang khác, phần thực thi sẽ tiếp tục trong khối **try** tại cại điểm xuất hiện lỗi ngoại lệ. Ngược lại, các handler ngoại lệ trong khối **except** được thực thi.

Như một thay thế cho cấp phát động, tiến trình có thể đơn giản cấp phát xác nhận vùng còn lại thay vì chỉ để dành chúng. Tuy nhiên việc cấp phát xác nhận như vậy lại tạo nên các khối nhớ không cần thiết đáng ra được sử dụng cho các tiến trình khác.

Trong ví dụ này, ta sử dụng hàm **VirtualFree** để giải phóng vùng nhớ đã xác nhận lẫn vùng nhớ để dành sau khi hoàn tất công việc. Hàm này được gọi hai lần: lần đầu để khử cấp phát các trang đã được cấp phát xác nhận, và lần sau để giải phóng toàn bộ các trang dưới dạng để dành.

```
#define PAGELIMIT 80
#define PAGESIZE 0x1000
INT PageFaultExceptionFilter(DWORD);
VOID MyErrorExit(LPTSTR);
```

```
LPTSTR lpNxtPage;
DWORD dwPages = 0;
VOID UseDynamicVirtualAlloc(VOID) {
    LPVOID lpvBase;
    LPTSTR lpPtr;
    BOOL bSuccess;
    DWORD i;
    /* Để dành các trang trong không gian địa chỉ ảo của tiến trình */
    lpvBase = VirtualAlloc(NULL, // hệ thống tự xác định địa chỉ
                           PAGELIMIT * PAGESIZE,// kích thước vùng cấp phát
                           MEM_RESERVE, // cấp phát dưới dạng để dành
                           PAGE_NOACCESS); // cách thức bảo vệ = không truy cập
    if (lpvBase == NULL)
        MyErrorExit("VirtualAlloc reserve");
    lpPtr = lpNxtPage = (LPTSTR) lpvBase;
    /* Sử dụng cấu trúc xử lý ngoại lệ try-exception để truy cập các trang.
       Nếu lỗi trang xuất hiện, bộ lọc ngoại lệ sẽ thực thi để cấp phát xác
       nhận các trang kế tiếp trong khối để dành */
    for (i = 0; i < PAGELIMIT*PAGESIZE; i++) {
        try {
            lpPtr[i] = 'a'; // Ghi vào bộ nhớ
            /* Nếu xuất hiện lỗi trang, cố gắng cấp phát xác nhận trang khác */
        }
        except (PageFaultExceptionFilter(GetExceptionCode()))
        {
            /* Đoạn này chỉ thực hiện khi hàm lọc không thể xác nhận
               trang kế tiếp */
            ExitProcess(GetLastError());
        }
    }
    /* Giải phóng các trang sau khi sử dụng. Đầu tiên là các trang đã
       được cấp phát xác nhận */
    bSuccess = VirtualFree(lpvBase, // địa chỉ cơ sở của kh'i nhớ
                           dwPages * PAGESIZE, // số byte các trang đã cấp phát
                           MEM_DECOMMIT); // hình thức là  khử xác nhận
    /* Cuối cùng, giải phóng toàn vùng nhớ (để dành) */
    if (bSuccess) {
        bSuccess = VirtualFree(lpvBase, // địa chỉ cơ sở của khối nhớ
                               0, // giải phóng to n khối nhớ
                               MEM_RELEASE); // giải phóng (hoàn toàn)
    }
}

INT PageFaultExceptionFilter(DWORD dwCode) {
    LPVOID lpvResult;
    /* Nếu xuất hiện lỗi ngoại lệ, thoát chương trình */
    if (dwCode != EXCEPTION_ACCESS_VIOLATION) {
        printf("exception code = %d\n", dwCode);
        return EXCEPTION_EXECUTE_HANDLER;
    }
    printf("page fault\n");
    /* Nếu các trang để dành đã được dùng thì thoát */
    if (dwPages >= PAGELIMIT) {
        printf("out of pages\n");
        return EXCEPTION_EXECUTE_HANDLER;
    }
    /* Ngược lại, cấp phát xác nhận một trang khác */
    lpvResult = VirtualAlloc((LPVOID) lpNxtPage, // cấp phát trang tiếp theo
                             PAGESIZE, // số byte kích thuóc trang
```

```
                                MEM_COMMIT, // cấp phát xác nhận các trang
                                PAGE_READWRITE); // truy cập đọc-ghi
    if (lpvResult == NULL) {
        printf("VirtualAlloc failed\n");
        return EXCEPTION_EXECUTE_HANDLER;
    }
    /* Tăng trang đếm, và chuyển lpNxtPage đến trang tiếp */
    dwPages++;
    lpNxtPage += PAGESIZE;
    /* Tiếp tục thực hiện nơi lỗi trang xuất hiện */
    return EXCEPTION_CONTINUE_EXECUTION;
}
```

Đoạn chương trình tiếp theo thực hiện thao tác tạo trang "lính canh". Trang này cung cấp cảnh báo khi truy cập các trang vùng nhớ. Điều này rất hữu ích cho các ứng dụng cần quản lý sự mở rộng của cấu trúc dữ liệu động.

Để tạo trang "lính canh", ta thiết lập cờ PAGE_GUARD trong hàm **VirtualAlloc**. Cờ này có thể dùng kết hợp với tất cả các cờ khác, trừ cờ PAGE_NOACCESS.

Nếu chương trình truy cập trang "lính canh", hệ thống sẽ phát sinh lỗi ngoại lệ STATUS_GUARD_PAGE (0x80000001). Hệ thống cũng xoá cờ PAGE_GUARD, loại bỏ tình trạng "lính canh" của trang vùng nhớ. Hệ thống sẽ không ngừng truy cập trang vùng nhớ với lỗi ngoại lệ STATUS_GUARD_PAGE.

Nếu một lỗi ngoại lệ xuất hiện trong suốt dịch vụ hệ thống, dịch vụ sẽ trả về giá trị xác định lỗi. Nếu sau đó ta truy cập lại trang này (mà chưa thiết lập lại tình trạng "lính canh"), thì sẽ không xảy ra lỗi ngoại lệ nữa.

Chương trình sau minh họa cách thực hiện của một trang lính canh, và hiện tượng xuất hiện lỗi dịch vụ hệ thống:

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
int main() {
    LPVOID lpvAddr;
    DWORD cbSize;
    BOOL vLock;
    LPVOID commit;
    cbSize = 512; // Vùng nhớ cần cấp phát.
    /* Gọi hàm cấp phát */
    lpvAddr = VirtualAlloc(NULL, cbSize, MEM_RESERVE, PAGE_NOACCESS);
    if (lpvAddr == NULL)
        fprintf(stdout, "VirtualAlloc failed on RESERVE with %ld\n",
                GetLastError());
    /* Cấp phát xác nhận vùng nhớ */
    commit = VirtualAlloc(NULL, cbSize, MEM_COMMIT, PAGE_READONLY|PAGE_GUARD);
    if (commit == NULL)
        fprintf(stderr, "VirtualAlloc failed on COMMIT with %ld\n",
                GetLastError());
    else
        fprintf(stderr, "Committed %lu bytes at address %lp\n", cbSize, commit);
    /* Khoá vùng nhớ đã xác nhận */
    vLock = VirtualLock(commit, cbSize);
    if (!vLock)
        fprintf(stderr, "Cannot lock at %lp, error = %lu\n", commit,
                GetLastError());
    else
        fprintf(stderr, "Lock Achieved at %lp\n", commit);
    /* Khoá vùng nhớ lần nữa */
    vLock = VirtualLock(commit, cbSize);
    if (!vLock)
        fprintf(stderr, "Cannot get 2nd lock at %lp, error = %lu\n", commit,
                GetLastError());
    else
```

```
        fprintf(stderr,"2nd Lock Achieved at %lp\n",commit);
}
```

Chương trình trên cho kết quả tương tự kết quả sau:

```
Committed 512 bytes at address 003F0000
Cannot lock at 003F0000, error = 0x80000001
2nd Lock Achieved at 003F0000
```

Chú ý: Lần khoá thứ nhất thất bại, tạo lỗi ngoại lệ STATUS_GUARD_PAGE. Tuy nhiên, trong lần khoá thứ hai, hàm thực hiện thành công, do hệ thống đã loại bỏ tình trạng "lính canh" của trang.

C. XỬ LÝ TẬP TIN

Tập tin là một đơn vị lưu trữ cơ bản để máy tính phân biệt các khối thông tin khác nhau, được lưu trữ trên các thiết bị lưu trữ phụ như là đĩa, băng từ, và được tổ chức theo các nhóm gọi là thư mục.

Để xử lý tập tin, ta có thể dùng các hàm trong C chuẩn như **fopen**, **fclose**, **fread**, **fwrite**, **fseek**, … trong môi trường Windows. Các hàm này được hỗ trợ trong thư viện **stdio.h**. Chúng ta sẽ không bàn về các hàm này ở đây.

Trong phần này, chúng ta sẽ tìm hiểu các hàm thao tác trên tập tin của Win32® cho phép các ứng dụng tạo, mở, cập nhật và xoá các tập tin, cũng như tìm hiểu các thông số hệ thống về tập tin.

1. Tạo và mở tập tin

Win32® API cung cấp hàm **CreateFile** để tạo một tập tin mới hoặc mở một tập tin đã có sẵn.

```
HANDLE CreateFile(LPCTSTR lpFileName, DWORD dwDesiredAccess, DWORD
dwShareMode, LPSECURITY_ATTRIBUTES lpSecurityAttributes, DWORD
dwCreationDisposition, DWORD dwFlagsAndAttributes, HANDLE hTemplateFile);
```

Trường lpFileName trỏ đến chuỗi ký tự zero xác định tên tập tin cần mở hoặc tạo. Trường dwDesiredAccess xác định cách thức truy cập đối tượng. Một ứng dụng có thể thực hiện truy cập đọc, ghi, đọc-ghi, sử dụng một hay kết hợp các giá trị sau:

Giá trị	Ý nghĩa
0	Xác định truy vấn thiết bị đến một đối tượng. Một ứng dụng có thể truy vấn thuộc tính thiết bị mà không cần phải truy cập thiết bị.
GENERIC_READ	Xác lập hình thức truy cập đọc. Dữ liệu có thể đọc từ tập tin, đồng thời dịch chuyển con trỏ tập tin. Để truy cập đọc-ghi, ta kết hợp với cờ GENERIC_WRITE.
GENERIC_WRITE	Xác lập hình thức truy cập ghi. Dữ liệu có thể được ghi vào tập tin, đồng thời dịch chuyển con trỏ. Để có thể truy cập đọc-ghi, ta kết hợp với cờ GENERIC_READ.

Bảng 7.4 Trường dwDesiredAccess xác định cách truy cập đối tượng

Trường dwShareMode thiết lập các bit cờ xác định cách chia sẻ đối tượng (tập tin). Nếu dwShareMode bằng 0, đối tượng không thể chia sẻ. Khi đó, ta không thể thao tác trên đối tượng cho đến khi đóng handle. Để chia sẻ đối tượng, ta kết hợp một trong các cờ sau:

FILE_SHARE_DELETE	Sử dụng trong Windows NT: Thao tác trên đối tượng chỉ thực hiện nếu yêu cầu truy cập xoá.
FILE_SHARE_READ	Thao tác trên đối tượng chỉ thực hiện nếu yêu cầu truy cập đọc.
FILE_SHARE_WRITE	Thao tác trên đối tượng chỉ thực hiện nếu yêu cầu truy cập ghi.

Bảng 7.5 Trường dwShareMode xác định cách chia sẻ đối tượng

Trường lpSecurityAttributes trỏ đến cấu trúc **SECURITY_ATTRIBUTES** xác định handle đối tượng có được chuyển cho các tiến trình con hay không. Ở đây chúng ta không dùng, và thiết lập giá trị là NULL.

Trường dwCreationDisposition xác lập thao tác tạo tập tin mới hay mở tập tin đã có. Dùng một trong các giá trị sau:

CREATE_NEW	Tạo mới một tập tin. Hàm này thất bại nếu tập tin đã có.
CREATE_ALWAYS	Tạo mới một tập tin. Nếu tập tin đã tồn tại, hàm sẽ tạo chồng lên, đồng thời xoá các thuộc tính hiện hành của tập tin.
OPEN_EXISTING	Mở một tập tin. Hàm thất bại nếu tập tin chưa có sẵn.
OPEN_ALWAYS	Mở một tập tin nếu có sẵn. Nếu tập tin chưa tồn tại, hàm sẽ tạo tập tin như sử dụng cờ CREATE_NEW.
TRUNCATE_EXISTING	Mở một tập tin. Khi mở, hệ thống khởi tạo kích thước tập tin lại về 0 byte. Tiến trình gọi cần mở tập tin ít nhất với dạng truy cập GENERIC_WRITE. Hàm thất bại nếu không tồn tại tập tin.

Bảng 7.6 Trường dwCreationDisposition xác lập thao tác tập tin

Trường dwFlagsAndAttributes xác định các thuộc tính và cờ cho tập tin. Ta có thể kết hợp các thuộc tính sau:

FILE_ATTRIBUTE_ARCHIVE	Tập tin archive. Ứng dụng dùng thuộc tính này để đánh dấu tập tin có thể sao lưu hoặc loại bỏ.
FILE_ATTRIBUTE_HIDDEN	Tập tin ẩn. Không hiển thị trong danh sách các tập tin thông thường trong các thư mục.
FILE_ATTRIBUTE_NORMAL	Tập tin không có thuộc tính nào khác. Thuộc tính này thường được dùng duy nhất.
FILE_ATTRIBUTE_OFFLINE	Dữ liệu tập tin không có sẵn. Dữ liệu được chỉ định di chuyển vật lý vào vùng lưu trữ offline.
FILE_ATTRIBUTE_READONLY	Tập tin chỉ đọc. Ứng dụng không thể ghi hoặc xoá dữ liệu trong tập tin.
FILE_ATTRIBUTE_SYSTEM	Tập tin là một phần của hệ điều hành, hoặc được sử dụng đặc biệt trong hệ thống.
FILE_ATTRIBUTE_TEMPORARY	Tập tin được dùng cho vùng lưu trữ tạm. Sau khi ứng dụng kết thúc, tập tin sẽ được xóa.

Bảng 7.7 Trường dwFlagsAndAttributes xác định các thuộc tính và cờ cho tập tin.

Các cờ xác định tập tin khá phức tạp, chúng ta không bàn kỹ ở đây. Trường cuối cùng là hTemplateFile xác định handle truy cập GENERAL_READ đến tập tin tạm. Tập tin tạm có vai trò hỗ trợ các thuộc tính tập tin và thuộc tính mở rộng cho tập tin được tạo. Trong Windows 95, giá trị hTemplateFile cần được gán bằng NULL.

Nếu thành công hàm trả về handle của tập tin xác định. Ngược lại, giá trị trả về là INVALID_HANDLE_VALUE.

Lưu ý, việc thiết lập giá trị dwDesiredAccess cho phép ứng dụng có thể truy vấn các thuộc tính thiết bị mà không thực sự truy cập thiết bị. Điều này rất hữu dụng, ví dụ trong trường hợp ứng dụng muốn xác định kích thước cùng các định dạng ổ đĩa mềm mà không cần phải có đĩa trong ổ đĩa.

Khi tạo một tập tin, hàm **CreateFile** thực hiện các chức năng sau:

- Kết hợp các cờ và thuộc tính tập tin được xác định bởi cờ dwFlagsAndAttributes với giá trị là FILE_ATTRIBUTE_ARCHIVE.
- Thiết lập kích thước tập tin bằng 0.
- Chép các thuộc tính mở rộng của tập tin tạm vào tập tin mới nếu biến hTemplateFile xác định.

Khi mở một tập tin có sẵn, hàm **CreateFile** thực hiện các chức năng sau:

- Kết hợp các cờ xác định bởi dwFlagsAndAttributes với các thuộc tính của tập tin hiện có. Hàm **CreateFile** sẽ bỏ qua các thuộc tính của tập tin xác định bởi cờ dwFlagsAndAttributes.
- Thiết lập kích thước tập tin dựa vào giá trị của dwCreationDisposition.
- Bỏ qua giá trị của biến hTemplateFile.

Nếu hàm tạo một tập tin trên ổ đĩa mềm không có đĩa mềm, hoặc trên CD-ROM không có đĩa CD, hệ thống sẽ đưa ra một hộp thoại thông điệp (message box) yêu cầu người dùng đưa đĩa mềm hoặc đĩa CD vào. Để hệ thống không thực hiện thao tác trên, cần thiết lập giá trị uMode trong hàm **SetErrorMode** là SEM_FAILCRITICALERRORS.

UINT SetErrorMode(**UINT** uMode);

Trong ví dụ sau, hàm **CreateFile** mở một tập tin đã có để đọc:

```
HANDLE hFile;
hFile = CreateFile("MYFILE.TXT", // mở tập tin MYFILE.TXT
GENERIC_READ, // mở để đọc
FILE_SHARE_READ, // chia sẻ để đọc
NULL, // không bảo mật
OPEN_EXISTING, // chỉ mở tập tin đã có
FILE_ATTRIBUTE_NORMAL, //Tập tin thường
NULL); // không có thộc tính tạm
if (hFile == INVALID_HANDLE_VALUE)
{
ErrorHandler("Could not open file."); // lỗi xử lý
}
```

Để xoá tập tin trên, trước hết ta đóng tập tin lại.

```
CloseHandle(hFile);
DeleteFile("MYFILE.TXT");
```

Trong ví dụ sau, hàm tạo một tập tin mới và mở ở chế độ ghi.

```
HANDLE hFile;
hFile = CreateFile("MYFILE.TXT", // tập tin MYFILE.TXT
GENERIC_WRITE, // tạo để ghi
0, // không chia sẻ
```

```
NULL, // không bảo mật
CREATE_ALWAYS, // ghi chồng nếu đã có
FILE_ATTRIBUTE_NORMAL | // tập tin bình thường
FILE_FLAG_OVERLAPPED, // không đồng bộ I/O
NULL); // không thuộc tính tạm
if (hFile == INVALID_HANDLE_VALUE)
{
ErrorHandler("Could not open file."); // lỗi xử lý
}
```

2. Tạo tập tin tạm

Các ứng dụng có thể nhận một tập tin duy nhất cho tập tin tạm bằng cách sử dụng hàm **GetTempFileName**. Để xác định đường dẫn đến thư mục chứa tập tin tạm được tạo, ta dùng hàm **GetTempPath**.

Hàm **GetTempFileName** tạo tên một tập tin tạm. Tên tập tin đầy đủ gồm đường dẫn nối với một chuỗi ký tự số thập lục phân thể hiện tên tập tin, và phần mở rộng là.TMP.

```
UINT GetTempFileName(LPCTSTR lpPathName, LPCTSTR lpPrefixString, UINT uUnique, LPTSTR lpTempFileName);
```

Trường lpPathName trỏ đến một chuỗi ký tự (kết thúc bằng ký tự NULL) xác định đường dẫn của tập tin, dùng các ký tự ANSI. Nếu trường này bằng NULL, hàm thất bại.

Trường lpPrefixString trỏ đến một chuỗi ký tự (kết thúc bằng ký tự NULL). Hàm sử dụng 3 ký tự đầu tiên của chuỗi như phần tiền tố của tập tin. Các ký tự sử dụng phải là ky tự ANSI.

Trường uUnique xác định một số nguyên không dấu (mà) hàm chuyển thành chuỗi ký tự thập lục phân sử dụng trong việc tạo tập tin tạm.

Trường lpTempFileName trỏ đến vùng nhớ đệm chứa tên tập tin tạm. Trường này là một chuỗi ký tự kết thúc NULL các ký tự ANSI. Độ dài vùng nhớ đệm được xác định bởi giá trị MAX_PATH của thư mục tương ứng.

Tập tin tạo được sẽ có dạng như sau:

path\preuuuu.TMP

Trong đó path là đường dẫn, xác định bởi giá trị lpPathName; pre là 3 ký tự đầu của chuỗi lpPrefixString; và uuuu là giá trị thập lục phân của uUnique.

Khi thoát khỏi hệ điều hành (tắt máy chẳng hạn), các tập tin tạm tạo bằng hàm này sẽ tự động bị xoá.

Để tránh các lỗi khi chuyển chuỗi ANSI, ứng dụng cần gọi hàm CreateFile trước để tạo tập tin tạm.

Nếu giá trị uUnique bằng 0, hàm **GetTempFileName** xác lập một con số duy nhất dựa trên thời điểm hiện tại của hệ thống. Nếu tập tin đã có, hệ thống tự tăng lên một số mới cho đến khi có một tên duy nhất.

Nếu thực hiện thành công, hàm trả về con số duy nhất xác định trong trường uUnique. Ngược lại, giá trị trả về là 0.

Để thu nhận đường dẫn tập tin tạm, ta dùng hàm **GetTempPath**.

DWORD GetTempPath(**DWORD** nBufferLength, **LPTSTR** lpBuffer);

Trường nBufferlength xác định kích thước vùng đệm chuỗi ký tự xác định bởi lpBuffer. Trường lpBuffer trỏ đến vùng đệm nhận chuỗi ký tự xác định đường dẫn tập tin tạm. Chuỗi ký tự kết thức bằng ký tự '\', ví dụ: C:\TEMP\.

Nếu thành công, hàm trả về độ lớn xác định kích thước chuỗi zero. Nếu giá trị trả về lớn hơn nBufferLength, giá trị trả về sẽ là kích thước vùng đệm cần để chứa đường dẫn. Ngược lại, giá trị trả về là 0 nếu hàm thất bại.

3. Sao chép và di chuyển tập tin

Để chép (*copy*) một tập tin, ta cần mở ở chế độ chỉ đọc. Sau đó dùng hàm **CopyFile** để chép vào một tập tin mới.

BOOL CopyFile(**LPCTSTR** lpExistingFileName, **LPCTSTR** lpNewFileName, **BOOL** bFailIfExists);

Trường lpExistingFileName và lpNewFileName trỏ đến chuỗi (kết thúc NULL) xác định tên tập tin đã có và tên tập tin mới. Trường bFialIfExists xác định cách tạo tập tin với tên mới trên. Nếu trường được thiết lập là TRUE, và tập tin có tên lpNewFileName đã tồn tại, hàm thất bại. Nếu trường được thiết lập là FALSE và tập tin đã tồn tại, hàm sẽ tạo tập tin mới chồng lên tập tin cũ.

Nếu thành công, hàm trả về giá trị khác 0. Ngược lại, giá trị trả về là 0.

Để di chuyển (*move*) một tập tin, trước hết cần phải đóng tập tin lại (nếu đang mở). Ta dùng hàm **MoveFile**. Hàm này thực hiện thao tác đổi tên một tập tin hay thư mục (bao gồm cả các tập tin con trong thư mục).

BOOL MoveFile(**LPCTSTR** lpExistingFileName, **LPCTSTR** lpNewFileName);

Hai trường trên lần lượt trỏ đến tên tập tin (thư mục) hiện có và tên tập tin (thư mục) mới. Tên tập tin (thư mục) mới cần phải chưa có trong đường dẫn của nó. Tên tập tin mới có thể trên một hệ thống hay ổ đĩa khác, trong khi tên thư mục mới phải cùng ổ đĩa với thư mục cũ.

Nếu thành công, giá trị trả về khác 0. Ngược lại, giá trị trả về là 0.

Ví dụ sau minh họa việc tạo và sử dụng tập tin tạm để copy một tập tin. Đầu tiên ứng dụng mở tập tin ORIGINAL.TXT bằng cách sử dụng hàm **CreateFile**. Sau đó ứng dụng sử dụng hàm **GetTempFileName** và **CreateFile** để tạo tập tin tạm. Ứng dụng đọc từng khối 4K dữ liệu vào vùng đệm, chuyển nội dung trong vùng đệm sang chữ hoa, và viết chúng xuống tập tin tạm. Sau khi chuyển toàn bộ tập tin trên sang tập tin tạm, ta đổi tập tin tạm thành ALLCAPS.TXT bằng cách dùng hàm **MoveFile**.

```
HANDLE hFile;
HANDLE hTempFile;
DWORD dwBytesRead, dwBytesWritten, dwPos;
char szTempName[MAX_PATH];
char buffer[4096];
/* Mở tập tin có sẵn */
hFile = CreateFile("ORIGINAL.TXT", // tên tập tin
                   GENERIC_READ, // mở để đọc
                   0, // không chia sẻ
                   NULL, // không bảo mật
                   OPEN_EXISTING, // tập tin đã có sẵn
                   FILE_ATTRIBUTE_NORMAL, // tập tin thông thường
                   NULL); // không thuộc tính tạm
if (hFile == INVALID_HANDLE_VALUE)
    ErrorHandler("Could not open file."); // xử lý lỗi
/* Tạo tập tin tạm */
GetTempFileName("\\TEMP", // thư mục chứa tập tin tạm
                "NEW", // phần đầu tập tin tạm
                0, // tạo tên duy nhất
                szTempName); // vùng đệm tên
hTempFile = CreateFile((LPTSTR) szTempName, // tên tập tin
                       GENERIC_READ | GENERIC_WRITE, // mở dạng đọc-ghi
                       0, // không chia sẻ
                       NULL, // không bảo mật
                       CREATE_ALWAYS, // tạo chồng nếu tập tin đã có
                       FILE_ATTRIBUTE_NORMAL, // tập tin thông thường
                       NULL); // không thuộc tính tạm
if (hTempFile == INVALID_HANDLE_VALUE)
    ErrorHandler("Could not create temporary file.");
/* Đọc khối 4K vào vùng đệm. Chuyển tất cả các ký tự trong vùng đệm sang dạng chữ
hoa. Viết vùng đệm vào tập tin tạm */
do {
    if (ReadFile(hFile, buffer, 4096, &dwBytesRead, NULL)) {
        CharUpperBuff(buffer, dwBytesRead);
        WriteFile(hTempFile, buffer, dwBytesRead, &dwBytesWritten, NULL);
    }
} while (dwBytesRead == 4096);
/* Đóng cả hai tập tin */
CloseHandle(hFile);
CloseHandle(hTempFile);
/* Chuyển tập tin tạm vào tập tin mới ALLCAPS.TXT */
if (!MoveFile(szTempName, "ALLCAPS.TXT"))
    ErrorHandler("Could not move temp. file.");
```

Hàm `CloseHandle` đóng một tập tin đang mở.

4. Đọc và ghi dữ liệu vào tập tin

Mỗi tập tin đang mở có một con trỏ tập tin xác định byte kế tiếp sẽ được đọc hoặc ghi. Khi một tập tin mở lần đầu tiên, hệ thống thiết lập con trỏ tập tin tại vị trí đầu tập tin. Mỗi khi đọc hoặc ghi vào một byte, con trỏ tập tin tự động dịch chuyển. Để thay đổi vị trí này, ta dùng hàm SetFilePointer.

```
DWORD SetFilePointer(HANDLE hFile, LONG lDistanceToMove,
      PLONG lpDistanceToMoveHigh, DWORD dwMoveMethod);
```

Trường hFile là handle của tập tin chứa con trỏ tập tin cần di chuyển. Handle này cần được tạo dưới dạng truy cập GENERIC_READ hoặc GENERIC_WRITE.

Trường lDistanceToMove và lpDistanceToMoveHigh xác định số byte con trỏ cần dịch chuyển. Nếu lpDistanceToMoveHigh bằng NULL, 32-bit thấp của lDistanceToMove xác định số byte cần di dời con trỏ. Ngược lại, hai trường trên thiết lập một giá trị 64-bit (không dấu) thể hiện số byte cần di dời con trỏ.

Trường dwMoveMethod xác định điểm gốc mà con trỏ cần di dời. Nếu trường này bằng FILE_BEGIN, điểm gốc là điểm đầu tiên (byte thứ 0) của tập tin. Nếu là FILE_CURRENT, điểm gốc là vị trí hiện tại của con trỏ. Và FILE_END xác định điểm gốc là vị trí cuối hiện tại của tập tin.

Nếu thành công, và lpDistanceToMoveHigh bằng NULL, giá trị trả về là DWORD thấp của con trỏ tập tin. Nếu lpDistanceToMoveHigh khác NULL, hàm trả về DWORD thấp của con trỏ tập tin, và trỏ trường này đến DWORD cao của con trỏ tập tin.

Nếu hàm thất bại và lpDistanceToMoveHigh bằng NULL, giá trị trả về là 0xFFFFFFFF. Để biết thêm thông tin lỗi, ta dùng hàm GetLastError. Nếu hàm trả về 0xFFFFFFFF và lpDistanceToMoveHigh, có thể hàm thành công hoặc thất bại, cần phải gọi hàm GetLastError để xác định. Đoạn chương trình sau trình bày vấn đề này:

```
/* Trường hợp 1: Gọi hàm với lpDistanceToMoveHigh == NULL */
/* Cố gắng di chuyển con trỏ tập tin của hFile một đoạn */
dwPtr = SetFilePointer(hFile, lDistance, NULL, FILE_BEGIN);
if (dwPtr == 0xFFFFFFFF) // Kiểm tra thất bại
{
    dwError = GetLastError(); // Nhận mã l—i
    ... // Xử lý lỗi
} // Cuối phần xử lý lỗi

/* Trường hợp 2: Gọi h m với lpDistanceToMoveHigh != NULL */
/* C' gắng di chuyển con trỏ tập tin hFile một 'oạn d i */
dwPtrLow = SetFilePointer(hFile, lDistLow, & lDistHigh, FILE_BEGIN);
/* Kiểm tra thất bại */
if (dwPtrLow==0xFFFFFFFF&&(dwError=GetLastError())!=NO_ERROR)
{
    // Xử lý l—i
    //...
} // Cuối lần xử lý lỗi
```

Để di chuyển tiến, ta thiết lập độ dịch chuyển một giá trị dương. Ngược lại, thiết lập giá trị âm để di chuyển lùi con trỏ tập tin. Nếu giá trị con trỏ tập tin sau khi dịch chuyển âm, hàm thất bại, và mã lỗi mà hàm GetlastError trả về là ERROR_NEGATIVE_SEEK.

Nếu muốn di chuyển con trỏ đến cuối tập tin để ghi tiếp, ta cũng có thể dùng hàm SetEndOfFile.

```
BOOL SetEndOfFile(HANDLE hFile);
```

Trường hFile là handle của tập tin cần di chuyển con trỏ đến cuối tập tin cần được tạo với dạng truy cập GENERAL_WRITE. Nếu thành công, hàm trả về giá trị khác 0. Ngược lại, giá trị trả về là 0.

Hàm ReadFile đọc dữ liệu từ một tập tin, từ điểm xác định bởi con trỏ tập tin. Sau khi đọc, con trỏ tập tin dịch chuyển một đoạn ứng với số byte thật sự đọc được. Tương tự, để ghi vào tập tin ta dùng hàm WriteFile.

```
BOOL ReadFile(HANDLE hFile, LPVOID lpBuffer, DWORD nNumberOfBytesToRead,
              LPDWORD lpNumberOfBytesRead, LPOVERLAPPED lpOverlapped);
BOOL WriteFile(HANDLE hFile, LPCVOID lpBuffer, DWORD nNumberOfBytesToWrite,
              LPDWORD lpNumberOfBytesWritten, LPOVERLAPPED lpOverlapped);
```

Handle tập tin hFile cần được tạo với dạng truy cập GENERAL_READ hoặc GENERAL_WRITE.

Trường lpBuffer trỏ đến vùng nhớ đệm nhận dữ liệu đọc từ tập tin, hay chứa dữ liệu cần ghi. Trường nNumberOfBytesToRead xác định số byte cần đọc. Trường lpNumberOfBytesRead trỏ đến

các byte đọc được. Hàm ReadFile thiết lập giá trị này bằng 0 trước khi thực hiện các thao tác khác để kiểm tra lỗi.

Trường nNumberOfBytesToWrite xác định số byte cần ghi. Nếu trường này bằng 0, hàm không ghi dữ liệu, nhưng vẫn được gọi thực hiện. Trường lpNumberOfBytesWritten xác định số byte ghi được, trường này cũng được hàm WriteFile thiết lập về 0 trước khi thực hiện các thao tác khác để kiểm tra lỗi.

Cuối cùng, trường lpOverlapped trỏ đến cấu trúc OVERLAPPED. Nếu tập tin được mở với cờ FILE_FLAG_OVERLAPPED, giá trị này phải khác NULL. Để đơn giản ta thiết lập giá trị bằng NULL, hàm sẽ đọc (ghi) tập tin từ vị trí con trỏ hiện tại và hàm chỉ trả kết quả về sau khi đọc (ghi) xong.

Hàm ReadFile sẽ dừng với một số lý do sau: thao tác ghi hoàn tất cuối đường ống ghi, đã đọc hết số byte yêu cầu, hoặc xảy ra lỗi khi đọc.

Nếu thành công, các hàm trả về giá trị khác 0. Ngược lại, giá trị trả về bằng 0.

5. Khoá và mở khoá tập tin

Mặc dù hệ thống cho phép nhiều ứng dụng có thể mở và ghi vào cùng một tập tin, các ứng dụng không nên thực hiện song song. Để ngăn chặn ứng dụng khác ghi vào phần dữ liệu của mình, ta có thể sử dụng hàm LockFile. Để mở khoá tập tin, cho phép các ứng dụng khác truy cập vùng dữ liệu, ta dùng hàm UnlockFile.

```
BOOL LockFile(HANDLE hFile, DWORD dwFileOffsetLow, DWORD dwFileOffsetHigh,
              DWORD nNumberOfBytesToLockLow, DWORD nNumberOfBytesToLockHigh);
BOOL UnlockFile(HANDLE hFile, DWORD dwFileOffsetLow, DWORD dwFileOffsetHigh,
              DWORD nNumberOfBytesToUnlockLow, DWORD nNumberOfBytesToUnlockHigh);
```

Trường hFile là handle của tập tin được mở dưới dạng GENERAL_READ hoặc GENERAL_WRITE, hoặc cả hai.

Trường dwFileOffsetLow và dwFileOffsetHigh xác định word thấp và cao của byte offset đầu tiên cần khoá hay mở khoá.

Trường nNumberofBytesToLockLow (nNumberofBytesToUnlockLow) và nNumberOfBytesToLockHigh (nNumberOfBytesToUnlockHigh) xác định word thấp và cao độ dài khoảng byte cần khoá hay mở khóa.

Nếu thành công, hàm trả về giá trị khác 0. Ngược lại, giá trị trả về là 0.

Ví dụ sau đây nối một tập tin vào một tập tin khác. Ứng dụng sử dụng hàm CreateFile mở tập tin ONE.TXT để đọc, và TWO.TXT để viết. Sau đó ứng dụng nối phần dữ liệu của tập tin ONE.TXT vào cuối tập tin TWO.TXT bằng cách đọc (dùng hàm ReadFile) và ghi (dùng hàm WriteFile) từng khối 4K dữ liệu. Trước khi viết vào tập tin thứ hai, ứng dụng dịch chuyển con trỏ đến cuối tập tin bằng cách dùng hàm SetFilePointer, sau đó khóa vùng cần ghi dùng hàm LockFile. Sau khi thực hiện thao tác ghi xong, ta mở khoá, dùng hàm UnlockFile, để các ứng dụng khác có thể sử dụng tập tin này.

```
HANDLE hFile;
HANDLE hAppend;
DWORD dwBytesRead, dwBytesWritten, dwPos;
char buff[4096];
/* Mở tập tin đã có */
hFile = CreateFile("ONE.TXT", // mở tập tin ONE.TXT
                   GENERIC_READ, // mở để đọc
                   0, // không chia sẻ
                   NULL, // không bảo mật
                   OPEN_EXISTING, // mở tập tin đã tồn tại
                   FILE_ATTRIBUTE_NORMAL, // tập tin bình thường
                   NULL); // không có thuộc tính tạm
if (hFile == INVALID_HANDLE_VALUE)
    ErrorHandler("Could not open ONE."); // xử lý lỗi
/* Mở tập tin đã có. Nếu chưa có, tạo tập tin mới */
hAppend = CreateFile("TWO.TXT", // mở tập tin TWO.TXT
                     GENERIC_WRITE, // mở để ghi
                     0, // không chia sẻ
                     NULL, // không bảo mật
                     OPEN_ALWAYS, // mở tập tin cũ hoặc tạo mới
                     FILE_ATTRIBUTE_NORMAL, // tập tin bình thường
                     NULL); // không có thuộc tính tạm
```

```
if (hAppend == INVALID_HANDLE_VALUE)
    ErrorHandler("Could not open TWO."); // xử lý lỗi
/* Nối tập tin thứ nhất vào cuối tập tin thứ hai. Khoá tập tin thứ hai để ngăn
   chặn các tiến trình khác truy cập khi đang ghi. Mở khoá sau khi ghi xong */
do {
   if (ReadFile(hFile, buff, 4096, &dwBytesRead, NULL)) {
       dwPos= SetFilePointer(hAppend, 0, NULL, FILE_END);
       LockFile(hAppend, dwPos, 0, dwPos+dwBytesRead, 0);
       WriteFile(hAppend, buff, dwBytesRead, &dwBytesWritten, NULL);
       UnlockFile(hAppend, dwPos, 0, dwPos+dwBytesRead, 0);
   }
} while (dwBytesRead == 4096);
/* Đóng cả hai tập tin */
CloseHandle(hFile);
CloseHandle(hAppend);
```

6. Đóng và xoá tập tin

Để sử dụng hiệu quả tài nguyên hệ thống, ứng dụng cần đóng các tập tin khi không cần dùng nữa bằng cách gọi hàm `CloseHandle`. Nếu ứng dụng bị ngắt và tập tin vẫn đang mở, hệ thống sẽ tự động đóng tập tin này.

```
BOOL CloseHandle(HANDLE hObject);
```

Nếu thành công, hàm trả về giá trị khác 0. Ngược lại, giá trị trả về là 0. Để biết các thông tin lỗi mở rộng, ta dùng hàm `GetLastError`.

Để xoá một tập tin, ta dùng hàm `DeleteFile`. Lưu ý rằng tập tin này cần phải đóng trước khi bị xóa.

```
BOOL DeleteFile(LPCTSTR lpFileName);
```

Trường `lpFileName` trỏ đến chuỗi (kết thúc bằng ký tự '\0') xác định tập tin cần xoá. Nếu thành công, hàm trả về khá trị khác 0. Ngược lại, giá trị trả về là 0. Hàm thất bại nếu tập nếu tập tin cần xoá không tồn tại. Trong Windows NT, hàm không thể xoá các tập tin đang mở. Tuy nhiên, trong Windows 95, tập tin đang mở vẫn có thể bị xoá.

7. Xử lý thư mục

Khi ứng dụng tạo một tập tin mới, hệ điều hành sẽ thêm tập tin này vào một thư mục xác định. Để tạo và xoá một thư mục, ta dùng hàm `CreateDirectory` và `RemoveDirectory`.

```
BOOL CreateDirectory(LPCTSTR lpPathName, LPSECURITY_ATTRIBUTES
     lpSecurityAttributes);
BOOL RemoveDirectory(LPCTSTR lpPathName);
```

Trường `lpPathName` trỏ đến chuỗi (kết thúc bằng ký tự '\0') xác định đường dẫn thư mục cần tạo (hay cần xoá). Kích thước chuỗi mặc định giới hạn cho đường dẫn là `MAX_PATH` ký tự. Trong trường hợp xoá thư mục, đường dẫn cần xác định thư mục là thư mục rỗng.

Trường `lpSecurityAttributes` trỏ đến cấu trúc `SECURITY_ATTRIBUTES` chứa trường lpSecurityDescriptor xác định mô tả mật của thư mục mới. Để đơn giản, ta gán giá trị `lpSecurityAttributes` bằng `NULL`, khi đó thư mục mới nhận các giá trị mô tả mật mặc định.

Nếu thành công, các hàm trả về giá trị khác 0. Ngược lại, giá trị trả về là 0.

Thư mục cuối của đường dẫn đang sử dụng gọi là thư mục hiện hành. Để xác định thư mục hiện hành, ứng dụng gọi hàm `GetCurrentDirectory`. Để thay đổi thư mục hiện hành, ứng dụng gọi hàm `SetCurrentDirectory`.

```
DWORD GetCurrentDirectory(DWORD nBufferLength, LPTSTR lpBuffer);
```

Trường nBufferLength xác định kích thước vùng đệm của chuỗi ký tự thể hiện thư mục hiện hành. Vùng đệm này phải dài đủ để chứa cả ký tự '\0' cuối chuỗi. Trường `lpBuffer` sẽ chứa chuỗi ký tự này.

Nếu thành công, hàm trả về giá trị xác định số ký tự được ghi vào vùng đệm, không chứa ký tự '\0'. Ngược lại, giá trị trả về là 0. Nếu vùng đệm lpBuffer không đủ lớn, giá trị trả về xác định kích thước cần thiết của vùng đệm, gồm cả byte `NULL` đáng dấu kết thúc chuỗi.

```
BOOL SetCurrentDirectory(LPCTSTR lpPathName);
```

Trường `lpPathName` trỏ đến chuỗi ký tự (kết thúc bằng '\0') xác định đường dẫn mới. Nếu thành công, hàm trả về giá trị khác 0. Ngược lại, giá trị trả về là 0.