Министерство образования и науки Российской Федерации

Кубанский государственный технологический университет

Кафедра вычислительной техники и АСУ

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ
Часть 1. Основы работы с WinAPI

Лабораторный практикум для студентов всех форм обучения специальности 230101 Вычислительные машины, комплексы, системы и сети

Краснодар
2011

Составители: канд. техн. наук, доц. А. Г. Мурлин;
ст. преп. А. Г. Волик;

УДК 681.31(031)

Системное программное обеспечение. Ч. 1: лабораторный практикум для студентов всех форм обучения специальности 230101 Вычислительные машины, комплексы, системы и сети / Сост.: А. Г. Мурлин, А. Г. Волик; Кубан. гос. технол. ун-т. Каф. вычислительной техники и АСУ. – Краснодар.: Изд. КубГТУ, 2011 – 96 с.

Содержат описания лабораторных работ, указания к их выполнению, задания и требования к оформлению отчета. Изложены основы разработки программ системного назначения для операционных систем семейства Windows. Рассмотрены вопросы работы с интерфейсом прикладного программирования WinAPI. Приведены исходные тексты примеров программ и рекомендуемая литература.

Ил. 14. Табл. 12. Библиогр.: 8 назв.

Печатается по решению методического совета Кубанского государственного технологического университета

Рецензенты:
зав. кафедрой ВТиАСУ КубГТУ, д-р техн. наук, проф. В. И. Ключко;
руководитель отдела телекоммуникаций ООО Информационный центр «Консультант», канд. техн. наук Н. Ф. Григорьев

Содержание

Введение

Методические указания содержат лабораторный практикум по дисциплине «Системное программное обеспечение» для специальности 230101 Вычислительные машины, комплексы, системы и сети.

Целью лабораторных работы является закрепление основ и углубление знаний в области устройства современных операционных систем семецства Windows с точки зрения программиста и порлучения практического опыта написания программ системного назанчения, а так же ознакомление студентов со средствами компиляции и отладки программ, которые предназначены для низкоуровневого взаимодействия с ОС.

При выполнении лабораторных работ должен соблюдаться следующий порядок выполнения работы:

- ознакомиться с описанием лабораторной работы;
- получить номер варианта задания у преподавателя;
- изучить необходимый теоретический материал, пользуясь настоящими указаниями и рекомендованной литературой;
- написать программу и отладить ее;
- подготовиться к ответам на теоретические вопросы по теме лабораторной работы;
- оформить отчет.

Все студенты должны предъявить индивидуальный отчет о результатах выполнения лабораторной работы. Допускается предъявление отчета в виде электронного документа.

Отчет должен содержать следующие пункты:

1) Титульный лист.
2) Краткое теоретическое описание.
3) Задание на лабораторную работу, включающее четкую формулировку задачи.
4) Листинг программы.
5) Результаты выполнения работы.

При сдаче отчета студент должен показать знание теоретического материала в объеме, определяемом тематикой лабораторной работы, а также пониманием сущности выполняемой работы.

Лабораторная работа № 1. Структура программы для Windows на языке C++

1 Цель работы

Цель работы – изучить структуру программы для операционной системы Windows. Научится создавать и регистрировать окно на основе класса окна. Изучить структуру и состав оконной процедуры.

2 Краткая теория

2.1 Первая программа для Windows

Расммотрим пример простейшей программы для Windows.

```
//HELLOWIN.C
#include <windows.h>
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR
szCmdLine, intiCmdShow)
{
   static char szAppName[] = "HelloWin";
   HWND hwnd;
   MSG  msg;

   WNDCLASSEX wndclass;
   wndclass.cbSize        = sizeof(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;
   wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION);

   RegisterClassEx (&wndclass);
   hwnd = CreateWindow (szAppName,  // имя класса окна
      "The Hello Program",          // заголовок окна
      WS_OVERLAPPEDWINDOW,          // стиль окна
      CW_USEDEFAULT,                // начальная позиция x
      CW_USEDEFAULT,                // начальная позиция y
      CW_USEDEFAULT,                // ширина окна
      CW_USEDEFAULT,                // высота окна
      NULL,                         //описатель родительского окна
      NULL,                         // описатель меню окна
      hInstance,                    // описатель экземпляра программы
      NULL) ;                       // параметры создания
   ShowWindow (hwnd, iCmdShow);
   UpdateWindow (hwnd);
```

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

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

    switch (iMsg)
    {
       case WM_PAINT:
          hdc = BeginPaint (hwnd, &ps);
          GetClientRect (hwnd, &rect);
          DrawText (hdc, "Hello, Windows!", -1, &rect, DT_SINGLELINE |
DT_CENTER | DT_VCENTER);
          EndPaint (hwnd, &ps);
          return 0;
       case WM_DESTROY:
          PostQuitMessage (0);
          return 0;
    }
    return DefWindowProc (hwnd, iMsg, wParam, lParam);
}
```

Программа создает обычное окно приложения. В окне, в центре рабочей области, выводится текст “Hello, Windows!”. При изменении размеров окна программа будет автоматически перемещать строку текста “Hello,Windows!” в новый центр рабочей области окна.

2.2 Настройки проекта

Windows поддерживает работу как с Юникод-строками, так и с однобайтными кодировками. Для этого в ее состав включены два набора функций: например существует две версии **DrawTextW** (Unicode) и **DrawTextA** (ANSI), а сама используемая в программе **DrawText**, по сути, является заглушкой для вызова одной из указанных выше функций**.**

При вызове функций WinAPI, в зависимости от настроек среды разработки и установленных флагов, вызывается либо Юникод-версия функции, либо обычная, однобайтная ANSI.

По-умолчанию современные версии сред разработки работают с Юникод-версиями функций, однако для простоты работы, далее в примерах подразумевается, что работа ведется с однобайтными кодировками.

Чтобы в настройках проекта указать какой тип строк используется по умолчанию необходимо во вкладке «General» выбрать для параметра «Character Set» значение «Use Multi-Byte Character Set» Вместо «Use Unicode character Set» (рисунок 1.1).

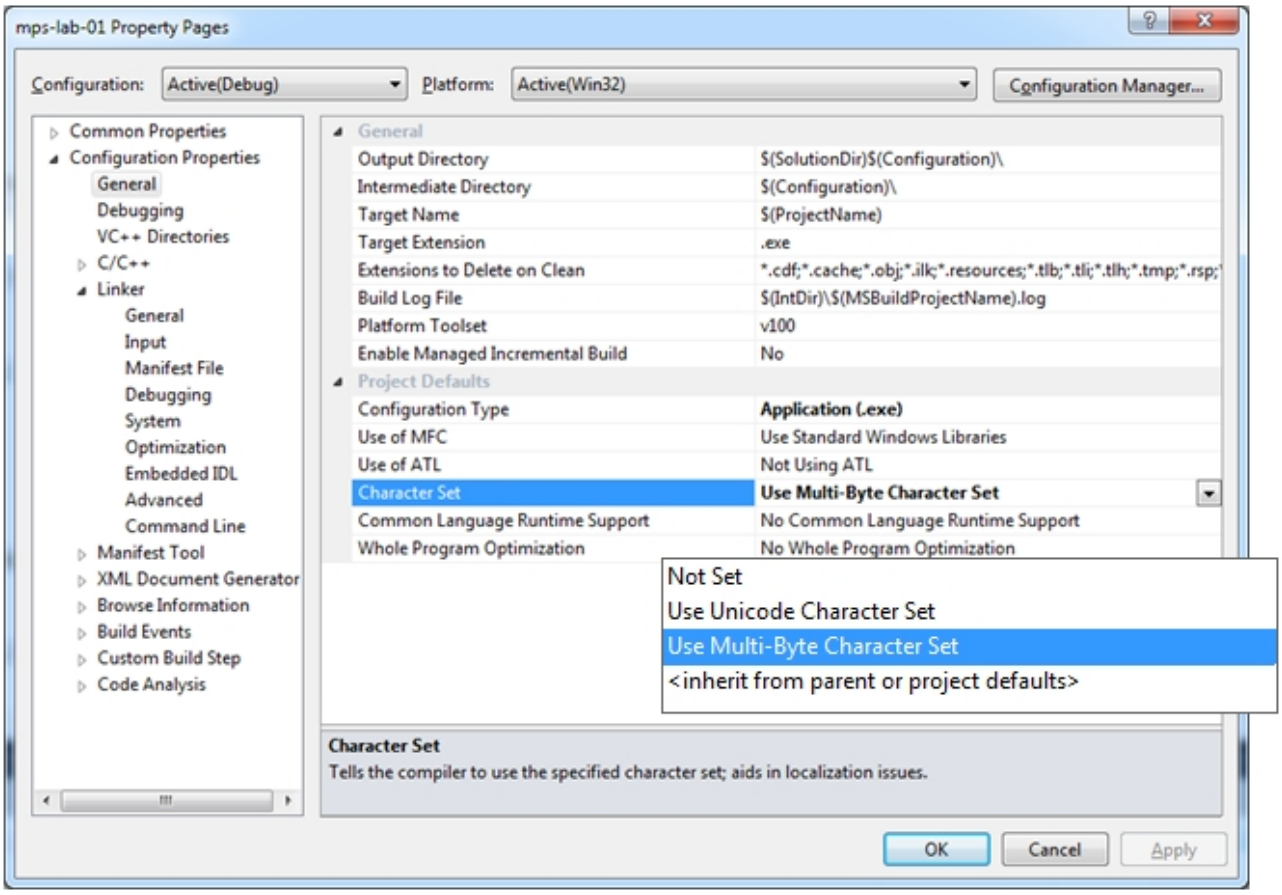

Рисунок 1.1 – Настроки среды разработки

2.3 Файл исходного текста программы на языке С

В файле имеется только две функции: WinMain и WndProc. WinMain – это точка входа в программу. WndProc –это “оконная процедура”. Каждое окно, не зависимо от того каким оно является, имеет свою оконную процедуру. Оконная процедура – это способ инкапсулирования кода, отвечающего за ввод информации (обычно с клавиатуры или мыши) и за вывод информации на экран. WndProc вызывается только из Windows. Однако в WinMain имеется ссылка на WndProc, поэтому эта функция описывается в самом начале программы, еще до определения WinMain.

2.3.1 Используемые функции Windows

HELLOWIN.C использует не менее 16-ти функций Windows. Здесь перечислены эти функции в порядке их появления в программе.

`LoadIcon` – загружает значок для использования в программе.

LoadCursor – загружает курсор мыши для использования в программе.

GetStockObject – получает графический объект (в данном случае кисть).

RegistrClassEx – регистрирует класс окна для определенного окна программы.

CreateWindow – создает окно на основе класса окна.

ShowWindow – выводит окно на экран.

UpdateWindow – заставляет окно перерисовать своё содержимое.

GetMessage – получает сообщение из очереди сообщений.

TranslateMessage – преобразует некоторые сообщения, полученные с помощью клавиатуры.

DispatchMessage – отправляет сообщение оконной процедуре.

BeginPaint – инициализирует начало процесса рисования окна.

GetClientRect – получает размер рабочей области окна.

DrawText – выводит текст.

EndPaint – прекращает рисование окна.

PostQuitMessage – вставляет сообщение “завершить” в очередь сообщений.

DefWindowProc – выполняет обработку сообщений по умолчанию.

2.3.2 Идентификаторы

Эти идентификаторы записаны в заголовочных файлах Windows. Некоторые из этих идентификаторов содержат двух или трёхбуквенный префикс, за которым следует символ подчеркивания: CS_HREDRAW, CS_VREDRAW, CW_USEDEFAULT, DT_CENTER, DT_SINGLELINE, DT_VCENTER, IDC_ARROW, IDI_APPLICATION, SND_ASYNC, SND_FILENAME, WM_DESTROY, WM_PAINT – это просто числовые константы. Префикс показывает основную категорию, к которой принадлежит константа, как показано в таблице 1.1.

Таблица 1.1 – Префиксы категорий идентификаиоров

Префикс	Категория
CS	Опция стиля класса
IDI	Идентификационный номер иконки
IDC	Идентификационный номер курсора
WS	Стиль окна
CW	Опция создания окна
WM	Сообщение окна
SND	Опция звука
DT	Опция рисования текста

2.3.3 Типы данных в приложениях Windows

UINT – Беззнаковое целое.

PSTR – Указатель на строку, т.е. char*.

WPARAM – Беззнаковое короткое целое.

LPARAM – 32-разрядное знаковое длинное целое.

Функция WndProc возвращает значение типа LRESULT. Оно определяется просто как LONG. Функция WinMain получает тип WINAPI, а функция WndProc получает тип CALLBACK. Оба эти идентификатора определяются как stdcall, что является ссылкой на особую последовательность вызовов функций, которая имеет место между самой ОС и ее приложением.

В HELLOWIN используется 4 структуры данных, определенных в заголовочных файлах Windows. Этими структурами являются:

Таблица 1.2 – Структуры WinAPI

Структура	Значение
MSG	Структура сообщения.
WNDCLASSEX	Структура класса окна.
PAINTSTRUCT	Структура рисования.
RECT	Структура прямоугольника.

Первые две структуры данных используются в WinMain для определения структур, названных msg и wndclass. Две вторые используются в WndProc для определения структур ps и rect.

Имеется еще идентификаторы написаных прописными буквами.

Таблица 1.3 – Структуры WinAPI

Идентификатор	Значение
HINSTANCE	Описатель экземпляра программы.
HWND	Описатель окна.
HDC	Описатель контекста устройства.
HICON	Описатель значка.
HCURSOR	Описатель курсора.
HBRUSH	Описатель кисти.

Описатель – это просто число, которое ссылается на объект (обычно длиной 32-бита).

2.4 Точка входа программы

Текст программы начинается с инструкции #include, которая позволяет включить в программу заголовочный файл WINDOWS.H. WINDOWS.H включает в себя много других заголовочных файлов, содержащих объявление функций, структур, новые типы данных и константы Windows. Затем следует объявление WndProc:

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

Это объявление необходимо потому, что внутри WinMain имеются ссылки на WndProc. Точкой входа программы для Windows является функция WinMain. WinMain всегда определяется следующим образом:

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

Эта функций использует последовательность вызовов WINAPI и, по своему завершению, возвращает ОС целое. В ней есть 4 параметра:

- Параметр hInstance называется описателем экземпляра. Это уникальное число, идентифицирующее программу, когда она работает под Windows.
- Параметр hPrewInstance – предыдущий экземпляр – в настоящее время устарел.
- Параметр szCmdLine – это указатель на строку оканчивающуюся нулем, в которой содержатся любые параметры, переданные программе из командной строки.
- Параметр iCmdShow – число, показывающее, каким должно быть выведено окно на экран в начальный момент. Это число задаётся при запуске программы другой программой.

Таблица 1.4 – Варианты значений параметра iCmdShow

Идентификатор	Назначение
SW_SHOWNORMAL	Вывести обычное окно
SW_HIDE	Скрывает окно и активизирует другое окно
SW_MINIMIZE	Минимизирует определенное окно и активизирует окно верхнего уровня в списке системы.
SW_MAXIMIZE	Максимизирует определенное окно.
SW_RESTORE	Активизирует и отображает окно. Если окно минимизировано или максимизировано, Windows восстанавливает его до первоначального размера и позиции (так же как SW_SHOWNORMAL).
SW_SHOW	Активизирует окно и отображает его в текущем размере и позиции.

SW_SHOWMINIMIZE	Активизирует окно и отображает его как минимизированное окно
SW_SHOWMAXIMIZE	Активизирует окно и отображает его как максимизируемое окно
SW_SHOWDEFAULT	Устанавливает состояние, основанное на флажке SW_, определенном в структуре STARTUPINFO, переданной к функции CreateProcess программой, которая начала приложение.
SW_SHOWNA	Отображает окно не активизируя его.
SW_SHOWNOACTIVE	Отображает окно в самом последнем размере и позиции, не активизируя его.
SW_SHOWMINNOACTIVE	Вывести свернутое окно, не активизируя его.

2.5 Регистрация класса окна

Окно всегда создаётся на основе класса окна. Класс окна идентифицирует оконную процедуру, которая выполняет процесс обработки сообщений, поступающих окну. На основе одного класса окна можно создать несколько окон. Перед созданием окна для вашей программы необходимо зарегистрировать класс окна путем вызова функции RegistrClassEx. Это расширенная версия функции RegistrClass из предыдущей версии Windows. У функции RegistrClassEx имеется один параметр: указатель на структуру типа WNDCLASSEX. Структура WNDCLASSEX определяется в заголовочных файлах Windows следующим образом:

```
typedef struct tagWNDCLASSEX
{
   UINT cbSize;
   UINT style;
   WNDPROC lpfnWndProc;
   int cbClsExtra;
   int cbWndExtra;
   HINSTANCE hInstance;
   HICON hIcon;
   HCURSOR hCursor;
   HBRUSH hbrBackground;
   LPCSTR lpszMenuName;
   LPCSTR lpszClassName;
   HICON hIconSm;
}
WNDCLASSEX;
```

В WinMain необходимо определить структуру типа WNDCLASSEX, обычно это делается так: WNDCLASSEX wndclass;

Затем задаются 12 полей структуры и вызывается RegisterClassEx:

```
RegisterClassEx(&wndclass);
```

Наиболее важными являются второе и третье от конца поля. Второе от конца поле является именем класса окна. Третье поле является адресом оконной процедуры, которая используется для всех окон, созданных на основе данного класса. Другие поля описывают характеристики всех окон, создаваемых на основе этого класса окна. Поле cbSize равно длине структуры. Инструкция:

```
wndclass.style = CS_HREDRAW | CS_VREDRAW;
```

осуществляет объединение двух идентификаторов стиля класса с помощью поразрядной операции OR. Эти два идентификатора показывают, что все окна, созданные на основе данного класса должны целиком перерисовываться при вертикальном или горизонтальном изменении размеров окна.

Таблица 1.5 – Таблица стилей класса

Стиль	**Описание**
CS_BYTEALIGNCLIENT	Выравнивание клиентской области окна на границе байта (по X) для расширения эффективность при выполнении операторов рисования.
CS_BYTEALIGNWINDOW	Выравнивает окно на границе байта (по X). Расширяет эффективность действий при изменении размеров и положения окна
CS_CLASSDC	Распределяет один контекст устройства между всеми окнами в классе.
CS_DBLCLKS	Посылает сообщение двойного щелчка оконной процедуре в том случае если двойной щелчок произошел в клиентской части
CS_GLOBALCLASS	Допускает, чтобы приложение создало окно класса независимо от значения hInstance, переданного функцией CreateWindow или CreateWindowEx
CS_HREDRAW	Перерисовка окна при изменении его ширины
CS_NOCLOSE	Отключает команду Close на Системном меню
CS_OWNDC	Распределяет уникальный контекст устройства для каждого окна в классе
CS_PARENTDC	Устанавливает область отсечения дочернего окна в родительском окне так, чтобы дочернее окно могло быть внутри родительского. Окно с CS_PARENTDC стилем получает регулярный контекст устройства от кэша системы контекстов устройства. Расширяет эффективность приложения.

CS_SAVEBITS	Сохраняет, как растр, блок отображаемого изображения, затененного окном. . Windows использует сохраненный растр, чтобы вновь создать отображаемое изображение, когда окно удалено. Windows отображает растр в первоначальном расположении и не посылает сообщения WM_PAINT окнам, затененным текущим окном, если память, используемая растром не была сброшена и если другие экранные действия не объявили неверным сохраненное изображение.
CS_VREDRAW	Перерисовка окна при изменении его высоты

Третье поле структуры WNDCLASSEX инициализируется с помощью инструкции: `wndclass.lpfnWndProc = WndProc;`

Эта инструкция устанавливает оконную `WndProc` как оконную процедуру данного окна. Она будет обрабатывать все сообщения всем окнам, созданным на основе данного класса окна. Следующие две инструкции:

```
wndclass.cbClsExtra = 0;
wndclass.cbWndExtra = 0;
```

резервируют некоторое дополнительное пространство в структуре класса и структуре окна. Программа может использовать это пространство по своему усмотрению.

В следующем поле находится описатель экземпляра класса:

```
wndclass.hInstance = hInstance;
```

Инструкции:

```
wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
wndclass.hIconSm = LoadIcon(NULL, IDI_APPLICATION);
```

устанавливают значок для всех окон созданных на основе данного класса окна.

Инструкция:

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

устанавливает курсор для всех окон созданных на основе данного класса окна.

Инструкция:

```
wndclass.hbrBackground = GetStockObject(WHITE_BRUSH);
```

устанавливает цвет фона окна.

Следующее поле задаёт меню класса окна. В приложении HELLOWIN меню отсутствует, поэтому поле установлено в NULL:

```
wndclass.lpszMenuName = NULL;
```

На последнем этапе классу присваивается имя:

```
wndclass.lpszClassName = szAppName;
```

После описания 12 полей вызывается функция RegisterClassEx для регистрации класса:

```
RegisterClassEx(&wndclass);
```

2.6 Создание окна

Класс окна определяет основные характеристики окна, что позволяет использовать один и тот же класс для создания множества различных окон. Вызов функции CreateWindow, фактически создает окно, таким образом, детализируется информация об окне. Вызов функции CreateWindow требует передачи информации об окне в качестве параметров. В HELLOWIN.C это выглядит так:

```
hwnd = CreateWindow (
szAppName, // имя класса окна
“The Hello Program”, // заголовок окна
WS_OVERLAPPEDWINDOW, // стиль окна
CW_USERDEFAULT, // начальное положение по x
CW_USERDEFAULT, // начальное положение по y
CW_USERDEFAULT, // начальный размер по x
CW_USERDEFAULT, // начальный размер по y
NULL, // описатель родительского окна
NULL, // описатель меню
hInstance, // описатель экземпляра класса
NULL); // параметры создания
```

Параметр с комментарием “имя класса окна” – szAppName содержит строку “HellloWin”, являющуюся именем только что зарегистрированного класса. Окно, созданное программой, является обычным перекрывающимся окном с заголовком, системным меню слева, иконками для сворачивания, закрытия и развертывания на весь экран. Это стандартный стиль окон, он называется WS_OVERLAPPEDWINDOW и помечен комментарием “стиль окна”. Комментарием “заголовок окна ” помечен текст выводимый в строке заголовка. Затем идут 4 параметра размеров и положения окна (идентификатор CW_USERDEFAULT позволяет установить все данные по умолчанию). Описатель родительского окна равен NULL т.к. у этого окна нет родительского. Описатель меню также равен NULL потому, что его

нет. И наконец, параметр создания равен NULL. При необходимости этот параметр используется в качестве указателя на какое ни будь данное. Вызов CreateWindow возвращает описатель созданного класса. Этот описатель хранится в переменной hwnd, которая имеет тип HWND. У каждого окна в Windows имеется свой описатель. В нашей программе описатель используется для того, чтобы ссылаться на окно.

2.7 Отображение окна

Для отображения окна нужно сделать еще два вызова. Первый из них:

```
ShowWindow( hwnd, iCmdShow );
```

Первым параметром является описатель окна созданный с помощью CreateWindow. Вторым параметром является величина, передаваемая функцией WinMain. Он задаёт вид окна на экране.

Второй вызов:

```
UpdateWindow (hwnd);
```

предназначен для перерисовки рабочей области.

2.8 Цикл обработки сообщений

Windows поддерживает очередь сообщений для каждой программы, работающей в данный момент в ОС. Когда происходит ввод информации, Windows преобразует ее в сообщение, которое помещается в очередь сообщений программы. Программа извлекает сообщение из очереди сообщений, выполняя блок команд, известный как “цикл обработки сообщений”:

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

Переменная msg – это структура типа MSG, которая определяется в заголовочных файлах Windows следующим образом:

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

Тип данных POINT – это тип данных другой структуры, которая определяется так:

```
typedef struct tagPOINT  {LONG x;LONG y;} POINT;
```

Вызов функции GetMessage, с которой начинается цикл обработки сообщений, извлекает сообщения из очереди сообщений: `GetMessage (&msg, NULL, 0, 0)` Этот вызов передаёт Windows указатель на структуру msg. Второй, третий. и четвертый параметры (0 и NULL) показывают, что программа получает сообщения от всех окон созданных ею. Поля структуры msg:

`hwnd` – описатель окна, для которого предназначено сообщение.

`message` – идентификатор сообщения. Это число которое идентифицирует сообщение. Для каждого сообщения существует свой идентификатор, который задаётся в заголовочном файле и начинается с префикса WM.

`wParam` – 32-разрядный параметр сообщения, смысл и значение которого зависят от самого сообщения.

`lParam` – другой 32-разрядный параметр сообщения, смысл и значение которого зависят от самого сообщения.

`pt` – координаты курсора мыши в момент помещения сообщения в очередь.

Если поле message сообщения не равно WN_QUIT, то функция GetMessage возвращает не нулевое значение. Сообщение WN_QUIT заставляет прервать цикл обработки сообщений. Инструкция: `TranslateMessage (&msg);` передаёт структуру msg обратно в Windows для преобразования какого либо сообщения от клавиатуры. Инструкция:

`DispatchMessage (&msg);` передаёт структуру msg обратно в Windows. Windows отправляет сообщение для его обработки соответствующей оконной процедуре – таким образом ОС вызывает оконную процедуру.

2.9 Оконная процедура

В нашей программе оконной процедурой является WndProc. Ей можно дать любое имя. В программе может содержаться более одной оконной процедуры. Оконная процедура всегда связана с определенным классом окна, который регистрируется с помощью RegisterClassEx. Функция CreateWindow создаёт окно на основе определенного класса окна. На основе одного и того же класса окна можно создать несколько окон. Оконная процедура всегда определяется следующим образом:

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

Первым параметром является hwnd, описатель полученного сообщения окна. Вторым параметром является 32-разрядное число, которое идентифицирует сообщение. Два последних параметра представляют дополнительную информацию о сообщении.

2.10 Обработка сообщений

В заголовочных файлах Windows определены идентификаторы, начинающиеся с префикса WM для каждого типа сообщений. Обычно используют конструкции switch и case для определения того, какое сообщение получила оконная процедура и то, как его обрабатывать. Если оконная процедура обрабатывает сообщение, то она должна вернуть 0, иначе если сообщение не обрабатывается, то оно должно передаваться функции `DefWindowProc`. Значение, возвращаемое функцией `DefWindowProc`, должно быть возвращаемым значением оконной процедуры.

В программе обрабатываются 3 сообщения: `WM_PAINT, WM_CREATE, WM_DESTROY`. Обработка сообщений в программе может выглядеть следующим образом:

```
switch (iMsg)
{
case WM_CREATE:
[обработка сообщения WM_CREATE];
return 0;
case WM_PAINT:
[обработка сообщения WM_PAINT];
return 0;
case WM_DESTROY:
[обработка сообщения WM_ DESTROY];
return 0;
}
return DefWindowProc (hwnd, iMsg, wParam, lParam,);
```

В этом коде функция `DefWindowProc` обрабатывает остальные сообщения по умолчанию.

2.11 Сообщение WM_PAINT

Сообщение WM_PAINT функции WndProc обрабатывается вторым. Это сообщение крайне важно при программировании по Windows. Она сообщает программе, что часть или вся рабочая область недействительна и ее надо перерисовать. При первом создании окна вся рабочая область недействительна, т.к. программа еще ни чего в ней не рисовала. Сообщение WM_PAINT заставляет программы что-то нарисовать в рабочей области. Обработка сообщения WM_PAINT почти всегда начинается с функции BeginPaint:

```
hdc = BeginPaint (hwnd, &ps);
```

```
и заканчивается вызовом функции EndPaint:
EndPaint (hwnd, &ps);
```

Второй параметр – это указатель на структуру PAINTSTRUCT. В этой структуре содержится некоторая информация, которую оконная процедура может использовать для рисования в рабочей области. При обработке вызова BeginPaint, Windows обновляет фон рабочей области с помощью кисти, заданной в поле hbrBackground структуры WNDCLASSEX, которая использовалась при регистрации класса окна. вызов BeginPaint делает всю рабочую область активной и возвращает описатель контекста устройства. Контекст устройства описывает устройство вывода информации и его драйвер. Описатель контекста устройства необходим для вывода в рабочую область текста и графики. Но с помощью этого описателя контекста устройства (полученного из BeginPaint) вне рабочей области ничего не нарисуете. Функция EndPaint освобождает описатель контекста устройства, после чего его нельзя использовать.

После того, как вызвали BeginPaint, вызывается GetClientRect:

GetClientRect (hwnd, &rect); Второй параметр – это указатель на переменную rect типа RECT. RECT – это структура прямоугольник, определенная в заголовочном файле Windows. Она имеет 4 поля типа LONG, имена полей: left, top, right, bottom. GetClientRect помещает в эти поля размеры рабочей области. Поля left и top всегда равны 0, а right и bottom равны соответственно ширине и высоте рабочей области в пикселях.

WndProc никак не использует структуру RECT, за исключением передачи указателя на нее в качестве 4 параметра функции DrawText:

```
DrawText (hdc, “Hello, Windows 95!”, -1, &rect, DT_SINGLELINE |
DT_CENTER | DT_VCENTER);
```

DrawText рисует текст. Третий параметр равный –1, определяет, что строка оканчивается нулевым символом. Последний параметр – это набор флагов, заданных в заголовочных файлах Windows. Флаги показывают, что текст выводится по центру, в одну строку, и внутри рабочей области.

Когда рабочая часть становится недействительной, WndProc получает новое сообщение WM_PAINT и снова рисует текст в центре рабочей области.

2.12 Сообщение WM_DESTROY

Это сообщение говорит о том, что Windows находится в процессе ликвидации окна в ответ на полученную от пользователя команду. Наша программа стандартно реагирует на это сообщение, вызывая:

```
PostQuitMessage (0);
```

Эта функция ставит сообщение WM_QUIT в очередь сообщений программы. Когда GetMessage получает сообщение WM_QUIT и возвращает 0, что заставляет WinMain прекратить обработку сообщений и закончить программу.

3 Контрольные вопросы

1) Какая функция является точкой входа в программу.
2) Какие параметры у функции WinMain и что они означают.
3) С помощью каких структур осуществляется регистрация класса окна (объясните назначение полей этих процедур).
4) Какая функция создаёт окно.
5) Для чего существует сообщение WM_PAINT.
6) Для чего существует сообщение WM_DESTROY.
7) Какая функция по умолчанию обрабатывает все сообщения.

4 Задание

1) Изучить описание лабораторной работы.
2) Составить простое приложение Windows, используя все возможные стили класса окна. Изменить координаты и размеры окна.
3) Создать приложение с различными опциями показа окна, изменяя параметр iCmdShow. Изменить заголовок окна и текст в окне.
4) Отладить и протестировать полученную программу.
5) Оформить отчёт.

Лабораторная работа № 2. Использование контекста устройства, вывод текста, использование полос прокрутки

1 Цель работы

Изучить возможности по выводу текстовой информации в окно с использованием полос прокрутки.

2 Краткая теория

В Windows можно выводить информацию только в рабочую часть окна. Windows – это ОС, управляющая сообщениями. Windows уведомляет приложения о различных событиях путем постановки синхронных сообщений в очередь сообщений приложения или путем отправки асинхронных сообщений соответствующей оконной процедуре (синхронные сообщения – это сообщения которые становятся в очередь сообщений, а асинхронные – это передающиеся на прямую без очереди). Посылая синхронное сообщение WM_PAINT, Windows уведомляет оконную процедуру о том, что часть рабочей области необходимо обновить.

2.1 Сообщение WM_PAINT

Большинство программ под Windows вызывают функцию UpdateWindow при инициализации в WinMain, сразу перед входом обработки сообщений. ОС использует эту возможность для асинхронной отправки в оконную процедуру первого сообщения WM_PAINT. Это сообщение информирует оконную процедуру о готовности рабочей области к рисованию. Оконная процедура получает сообщение WM_PAINT при возникновении следующих ситуаций:

- Предварительно скрытая часть окна открылась, когда пользователь передвинул окно или совершил какое-то действие, в результате которого окно опять стало видимым;
- Пользователь изменил размеры окна;
- В программе для прокрутки части рабочей области используются функции ScrollWindow и ScrollDC;
- Для генерации сообщения WM_PAINT в программе используются функции InvalidateRect или InvalidateRgn.
- Windows посылает синхронно сообщение WM_PAINT в следующих случаях:
- Удаляется окно диалога или окно сообщения, которое перекрывало часть рабочей области;
- Раскрывается горизонтальное меню и затем удаляется с экрана;
- В некоторых случаях Windows всегда сохраняет перекрываемую область:
- При перемещении курсора мыши.
- Перемещение иконки по рабочей области.

2.2 Действительные и недействительные прямоугольники

Часто полностью перерисовывать рабочую область не нужно, а лишь ее часть. Например, когда часть рабочей области закрыто окном диалога. Перерисовка требуется только для прямоугольной области, вновь открывающейся при удалении окна диалога. Эта область называется “недействительным регионом” или ”регионом обновления”. Появление недействительного региона вынуждает Windows поместить синхронное сообщение WM_PAINT в очередь сообщений программы.

В Windows для каждого окна поддерживается структура информации о рисовании. В этой структуре содержатся координаты минимально возможного прямоугольника, содержащего недействительную область. Эта информация носит название недействительного прямоугольника. Если еще один регион становится недействительным, то ОС рассчитывает новый недействительный прямоугольник, который содержит оба регион.

Оконная процедура, вызывая функцию InvalidateRect, может задать недействительный прямоугольник в своей рабочей области. Если в очереди сообщений содержится WM_PAINT, то ОС рассчитывает новый недействительный прямоугольник, иначе помещает его в очередь сообщений. При получении сообщения WM_PAINT, оконная процедура получит и координаты недействительного прямоугольника. Однако, координаты можно получить и с помощью функции GetUpdateRect, не дожидаясь сообщения WM_PAINT.

После того как оконная процедура вызывает функцию BeginPaint при обработке сообщения WM_PAINT, вся рабочая область становится действительной. Программа, вызвав функцию ValidateRect, также может сделать действительной любую прямоугольную зону.

2.3 Контекст устройства

Описатель – это просто число, которое ОС использует для внутренней ссылки на объект. После получения описателя, его используют в различных графических функциях. Описатель контекста устройства – это паспорт окна для функций GDI (графический интерфейс устройства). Описатель даёт возможность использовать функции GDI при выводе графики.

Контекст устройства фактически является структурой данных, которая внутренне поддерживается GDI. Контекст устройства связан с конкретным устройством вывода информации, таким как принтер, плоттер или дисплей. Что касается дисплея, то в данном случае контекст устройства связан с конкретным окном на экране.

Когда программе необходимо начать рисовать, она должна получить описатель контекста устройства. После окончания рисования его необходимо освободить. Когда программа освобождает описатель, он становится недействительным и не может далее использоваться.

2.4 Получение описателя контекста устройства

Для получения описателя контекста устройства обычно используют два способа.

2.4.1 Получение описателя контекста устройства при помощи BeginPaint

Применяется при обработке сообщения WM_PAINT. Применяются две функции: BeginPaint и EndPaint. Для этих функций требуется описатель окна и адрес переменной типа структуры PAINTSTRUCT. Переменная этого типа определяется внутри оконной процедуры следующим образом: PAINTSTRUCT ps;

Переменная описателя контекста устройства определяется следующим образом: HDC hdc;

Тип HDC определяется как 32-разрядное беззнаковое целое.

Типовой процесс обработки сообщения WM_PAINT:

```
case WM_PAINT:
hdc = BeginPaint ( hwnd, &ps ); // начало рисования
[ использование функций GDI ]
EndPaint ( hwnd, &ps ); // конец рисования
return 0;
```

Функции BeginPaint и EndPaint должны вызываться парой. Если сообщение WM_PAINT не обрабатывается, то оно передаётся в DefWindowProc. DefWindowProc обрабатывает сообщение WM_PAINT следующим образом:

```
case WM_PAINT:
hdc = BeginPaint ( hwnd, &ps ); // начало рисования
EndPaint ( hwnd, &ps ); // конец рисования
return 0;
```

Делает неактивную область активной, но не перерисовывает её.

2.4.2 Структура информации о рисовании

Структура информации о рисовании – это PAINTSTRUCT. Она определяется так:

```
typedef struct tagPAINTSTRUCT
{
HDC hdc;
BOOL fErase;
RECT rcPaint;
BOOL fRestore;
BOOL fIncUpdate;
BYTE rgbReserved[32];
} PAINTSTRUCT;
```

Windows заполняет поля структуры, когда программа вызывает функцию BeginPaint. Программе можно использовать только 3 первых поля, остальными пользуется ОС. Поле hdc – это описатель контекста устройства. В подавляющем большинстве случаев, в поле fErase установлен флаг TRUE, означающий, что Windows обновит фон недействительного прямоугольника. Windows перерисует фон, используя кисть, заданную в поле hbrBackground структуры WNDCLASSEX, которая использована при регистрации класса окна во время инициализации WinMain. Многие программы для Windows используют белую кисть: wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;

Однако, если, вызывается функция InvalidateRect, то делается недействительным прямоугольник рабочей зоны программы, а последний параметр этой функции определяет, необходимо ли стирать фон. Если параметр равен FALSE, Windows не будет стирать фон и поле fErase также будет равно FALSE.

Поле rcPaint структуры PAINTSTRUCT определяет границы недействительного прямоугольника. Недействительный прямоугольник – это та область, которую необходимо перерисовать.

Прямоугольник rcPaint в PAINTSTRUCT – это не только недействительный прямоугольник, это также и отсекающий прямоугольник. Это означает, что Windows рисует только внутри отсекающего прямоугольника (т. е. если недействительная зона не является прямоугольником, Windows рисует только внутри этой зоны). Когда используется описатель контекста устройства из структуры PAINTSTRUCT, Windows не будет рисовать вне прямоугольника rcPaint.

Чтобы при обработке сообщения WM_PAINT рисовать вне прямоугольника rcPaint, необходимо сделать вызов: InvalidateRect (hWnd, NULL, TRUE) ;

перед вызовом BeginPaint. Это сделает недействительной всю рабочую область и обновит ее фон. Если же значение последнего параметра будет равно FALSE, то фон обновляться не будет. Все что там было, останется неизменным.

2.4.3 Получение описателя контекста устройства при помощи GetDC

Вызывайте GetDC для получения описателя контекста устройства и ReleaseDC, если он больше не нужен:

```
hdc = GetDC (hwnd) ;
[использование функций GDI]
ReleaseDC (hwnd, hdc) ;
```

Также как BeginPaint и EndPain, функции GetDC и ReleaseDC следует вызывать парой. В отличии от описателя контекста устройства, полученного из структуры PAINTSTRUCT, в описателе контекста устройства, возвращаемом функцией GetDC, определен отсекающий прямоугольник, равный всей рабочей области. Теперь можно рисовать во всей рабочей области.

2.5 Функция TextOut

Рассмотрим функцию TextOut: TextOut (hdc, x, y, psString, iLength) ; Первый параметр – это описатель контекста устройства, являющийся возвращаемым значением функций GetDC и BeginPain.

Цвет фона текста не является цветом фона, который установлен при определении класса окна. Фон в классе окна – это кисть, являющаяся шаблоном, которая может иметь, а может и не иметь чистый цвет, и которую Windows использует для закрашивания рабочей области. Поэтому в начале программы следует задавать цвет кисти, к примеру WHITE_BRUSH.

Параметр psString – это указатель на символьную строку, а iLength – длина строки. TextOut не определяет конца строки по нулевому символу. Значения x и y определяет точку начала строки.

Задаваемый по умолчанию режим отображения называется MM_TEXT. В режиме отображения MM_TEXT логические единицы соответствуют физическим, которыми являются пиксели, задаваемые относительно левого верхнего угла рабочей области.

2.6 Системный шрифт

Контекст устройства также определяет шрифт, который Windows использует при выводе текста в рабочую область. По умолчанию задается системный шрифт или SYSTEM_FONT. Системный шрифт – это шрифт, который Windows использует для текста заголовка, меню и окон диалога. Системный шрифт является растровым шрифтом, т. е. все символы определены как пиксельные шаблоны. Windows имеет несколько растровых шрифтов с разными размерами (для работы с разными видеоадаптерами).

2.7 Размер символа

Размер символа можно получить с помощью вызова функции GetTextMetrics. Для функции GetTextMetrics требуется описатель контекста устройства, поскольку ее возвращаемым значением является информация о шрифте, выбранным в данное время в контексте устройства. Windows копирует различные значения метрических параметров текста в структуру типа TEXTMETRICS. Для использования функции GetTextMetrics, во-первых, необходимо определить переменную: TEXTMETRICS tm; Далее нужно получить описатель контекста устройства и вызвать GetTextMetrics:

```
hdc = GetDC (hwnd) ;
GetTextMetrics (hdc, &tm) ;
ReleaseDC(hwnd, hdc);
```

2.8 Метрические параметры текста

Структура TEXTMETRICS обеспечивает полную информацию о выбранном в контексте устройства шрифте. Значение tmInternalLeading – это величина пустого пространства, отведенного для указания специальных знаков над символом. Если это значение равно 0, то помеченные прописные буквы делаются немного меньше, чтобы специальный символ поместился внутри верхней части символа. Значение tmExternalLeading – это величина пустого пространства, которое разработчик шрифта установил для использования между строками символов. В структуре TEXTMETRICS имеется два поля, описывающих ширину символа:

tmAveCharWidth – усредненная ширина символов строки,

tmMaxCharWidth – ширина самого широкого символа шрифта.

Для фиксированного шрифта эти величины одинаковы.

2.9 Форматирование текста

Поскольку размеры системного шрифта не меняются в рамках одного сеанса работы с Windows, необходимо вызывать GetTextMetrics только один раз. Хорошо делать это при обработке сообщения WM_CREATE. Это сообщение первое, которое принимает оконная процедура. Windows вызывает оконную процедуру с сообщением WM_CREATE, когда вызывается в WinMain функция CreateWindow.

Внутри оконной процедуры можно определить две переменные для хранения средней длины символов (cxChar) и полной высоты символов (cyChar):

```
static int cxChar, cyChar;
```

Рассмотрим пример обработки сообщения WM_CREATE:

```
case WM_CREATE:
hdc = GetDC (hwnd) ;
GetTextMetrics (hdc, &tm) ;
cxChar = tm.tmAveCharWidth ;
cyChar = tm.tmHeigth + tm.tmExternalLeading ;
ReleaseDC (hwnd, hdc) ;
return 0 ;
```

Если нет нужды учитывать межстрочное пространство, то можно использовать: cyChar = tm.tmHeigth ;

Обычно оставляют пустое поле cyChar в верхней части рабочей области, и cxChar в ее левой части. Для вывода на экран нескольких строк текста с выравниваем па левому краю при вызове функции TextOut используют следующие значение координаты x: cxChar значение координаты y в TextOut: cyChar*(1+i), где i – номер строки начиная с 0.

Программа, которая выводит на экран результаты полученные при вызове функции GetSystemMetrics, приведена в качестве примера.

Пример 2.1 – Файл Sismets.h

```
#define NUMLINES ((int) (sizeof sysmetrics / sizeof sysmetrics [0]))
struct {
   int  iIndex ;
   char *szLabel ;
   char *szDesc ;
}
sysmetrics [] ={
SM_CXSCREEN,      "SM_CXSCREEN",     "Screen width in pixels",
SM_CYSCREEN,      "SM_CYSCREEN",     "Screen height in pixels",
SM_CXVSCROLL,     "SM_CXVSCROLL",    "Vertical scroll arrow width",
SM_CYHSCROLL,     "SM_CYHSCROLL",    "Horizontal scroll arrow height",
SM_CYCAPTION,     "SM_CYCAPTION",    "Caption bar height",
SM_CXBORDER,      "SM_CXBORDER",     "Window border width",
SM_CYBORDER,      "SM_CYBORDER",     "Window border height",
SM_CXDLGFRAME,    "SM_CXDLGFRAME",   "Dialog window frame width",
SM_CYDLGFRAME,    "SM_CYDLGFRAME",   "Dialog window frame height",
SM_CYVTHUMB,      "SM_CYVTHUMB",     "Vertical scroll thumb height",
SM_CXHTHUMB,      "SM_CXHTHUMB",     "Horizontal scroll thumb width",
SM_CXICON,        "SM_CXICON",       "Icon width",
SM_CYICON,        "SM_CYICON",       "Icon height",
SM_CXCURSOR,      "SM_CXCURSOR",     "Cursor width",
SM_CYCURSOR,      "SM_CYCURSOR",     "Cursor height",
SM_CYMENU,        "SM_CYMENU",       "Menu bar height",
SM_CXFULLSCREEN,  "SM_CXFULLSCREEN", "Full screen client area width",
SM_CYFULLSCREEN,  "SM_CYFULLSCREEN", "Full screen client area height",
SM_CYKANJIWINDOW, "SM_CYKANJIWINDOW", "Kanji window height",
SM_MOUSEPRESENT,  "SM_MOUSEPRESENT", "Mouse present flag",
SM_CYVSCROLL,     "SM_CYVSCROLL",    "Vertical scroll arrow height",
SM_CXHSCROLL,     "SM_CXHSCROLL",    "Horizontal scroll arrow width",
SM_DEBUG,         "SM_DEBUG",        "Debug version flag",
SM_SWAPBUTTON,    "SM_SWAPBUTTON",   "Mouse buttons swapped flag",
SM_RESERVED1,     "SM_RESERVED1",    "Reserved",
SM_RESERVED2,     "SM_RESERVED2",    "Reserved",
SM_RESERVED3,     "SM_RESERVED3",    "Reserved",
SM_RESERVED4,     "SM_RESERVED4",    "Reserved",
SM_CXMIN,         "SM_CXMIN",        "Minimum window width",
SM_CYMIN,         "SM_CYMIN",        "Minimum window height",
SM_CXSIZE,        "SM_CXSIZE",       "Minimize/Maximize icon width",
SM_CYSIZE,        "SM_CYSIZE",       "Minimize/Maximize icon height",
SM_CXFRAME,       "SM_CXFRAME",      "Window frame width",
SM_CYFRAME,       "SM_CYFRAME",      "Window frame height",
SM_CXMINTRACK,    "SM_CXMINTRACK",   "Minimum window tracking width",
SM_CYMINTRACK,    "SM_CYMINTRACK",   "Minimum window tracking height",
SM_CXDOUBLECLK,   "SM_CXDOUBLECLK",  "Double click x tolerance",
SM_CYDOUBLECLK,   "SM_CYDOUBLECLK",  "Double click y tolerance",
SM_CXICONSPACING, "SM_CXICONSPACING","Horizontal icon spacing",
SM_CYICONSPACING, "SM_CYICONSPACING","Vertical icon spacing",
SM_MENUDROPALIGNMENT,"SM_MENUDROPALIGNMENT","Left or right menu drop",
SM_PENWINDOWS,    "SM_PENWINDOWS",   "Pen extensions installed",
```

```
SM_DBCSENABLED,   "SM_DBCSENABLED",  "Double-Byte Char Set enabled",
SM_CMOUSEBUTTONS, "SM_CMOUSEBUTTONS","Number of mouse buttons",
SM_SHOWSOUNDS,    "SM_SHOWSOUNDS",   "Present sounds visually"
} ;
```

Пример 2.2 – Файл Sismets1.c

```
#include <windows.h>
#include <string.h>
#include "sysmets.h"
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "SysMets1" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;
     wndclass.cbSize        = sizeof (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 ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;
     RegisterClassEx (&wndclass) ;
     hwnd = CreateWindow (szAppName, "Get System Metrics No. 1",
                          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 iMsg, WPARAM wParam, LPARAM
lParam)
     {
     static int  cxChar, cxCaps, cyChar ;
     char        szBuffer[10] ;
     HDC         hdc ;
     int         i ;
     PAINTSTRUCT ps ;
     TEXTMETRIC  tm ;
     switch (iMsg)
          {
```

```
        case WM_CREATE :
             hdc = GetDC (hwnd) ;
             GetTextMetrics (hdc, &tm) ;
             cxChar = tm.tmAveCharWidth ;
             cxCaps = (tm.tmPitchAndFamily & 1 ? 3:2) * cxChar / 2;
             cyChar = tm.tmHeight + tm.tmExternalLeading ;
             ReleaseDC (hwnd, hdc) ;
             return 0 ;
        case WM_PAINT :
             hdc = BeginPaint (hwnd, &ps) ;
             for (i = 0 ; i < NUMLINES ; i++) {
                 TextOut (hdc, cxChar, cyChar * (1 + i),
                    sysmetrics[i].szLabel,
                    strlen (sysmetrics[i].szLabel)) ;
                 TextOut (hdc, cxChar + 22 * cxCaps,
                    cyChar * (1 + i), sysmetrics[i].szDesc,
                    strlen (sysmetrics[i].szDesc)) ;
                 SetTextAlign (hdc, TA_RIGHT | TA_TOP) ;
                 TextOut (hdc, cxChar + 22 * cxCaps + 40 * cxChar,
                    cyChar * (1 + i), szBuffer,
                    wsprintf (szBuffer, "%5d",
                    GetSystemMetrics (sysmetrics[i].iIndex)));
                 SetTextAlign (hdc, TA_LEFT | TA_TOP) ;
             }
             EndPaint (hwnd, &ps) ;
             return 0 ;
        case WM_DESTROY :
             PostQuitMessage (0) ;
             return 0 ;
        }
    return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
    }
```

2.10 Оконная процедура программы

Оконная процедура WndProc обрабатывает в программе 3 сообщения: WM_CREATE, WM_PAINT, WM_DESTROY. При обработке сообщения WM_CREATE SYSMETS1 получает контекст устройства путем вызова GetDC, а также размеры текста вызывая функцию GetTextMetrics. SYSMETS1 также сохраняет среднюю ширину символов верхнего регистра в статической переменной cxCaps. Для фиксированного шрифта cxCaps = cxChar. Для пропорционального шрифта cxCaps = 150% от cxChar. Младший бит поля tmPitchAndFamily структуры TEXTMETRICS равен 1 для пропорционального шрифта и 0 для фиксированного. SYSMETS1 использует значение этого бита следующим образом: cxCaps = (tm.tmPitchAndFamily & 1 ? 3 : 2)*cxChar/2 ;

SYSMETS1 полностью выполняет процедуру рисования окна во время обработки сообщения WM_PAINT. В цикле for обрабатываются все элементы структуры sysmetrics, определенной в SYSMETS.H. Три колонки

текста выводятся на экран тремя функциями TextOut. В каждом случае третий параметр TextOut – это выражение: cxChar * (1 + i).

Этот параметр указывает в пикселях положение верхней границы строки символов относительно верхней границы рабочей области. Первая инструкция TextOut выводит на экран первую колонку идентификаторов, написанных прописными буквами. Второй параметр TextOut – это cxChar. Он делает отступ от левого края равным одному символу. Текст берется из поля szLabel структуры sysmetrics. Функция strlen используется для получения длины строки.

Вторая инструкция TextOut выводит на экран описание значений системных размеров. Эти описания хранятся в поле szDesc структуры sismetrics. Третий параметр в этом случае у TextOut имеет следующий вид: cxChar + 22 * cxCaps. Максимальная длина первой колонки равна 20 символам, поэтому вторая колонка должна начинаться как минимум на 20 * cxCaps правее начала первой колонки.

Треть инструкция TextOut выводит на экран численные значения, полученные от функции GetTextMetrics. Пропорциональный шрифт делает форматирование колонки, выровненных по правому краю чисел, слегка обманчивым. Каждая цифра от 0 до 9 имеет одну и ту же ширину, но эта ширина больше, чем ширина пробела. Числа могут быть шириной в одну и более цифр, поэтому начальное горизонтальное положение чисел может меняться от строки к строке.

Функция SetTextAlign позволяет выводить колонку выровненных по правому краю чисел, задавая положение последней цифры, вместо первой.

```
SetTextAlign (hdc, TA_RIGHT | TA_TOP);
```

После этого координаты, переданные TextOut, будут задавать правый верхний угол строки текста вместо ее левого верхнего угла.

2.11 Размер рабочей области

Большинство окон можно развернуть, и рабочая область займет весь экран, за исключением панели заголовка программы. Полный размер этой рабочей области становится доступным, благодаря функции GetSystemMetrics, использующей параметры SM_CXFULLSCREEN и SM_CYFULLSCREEN. Минимальный размер окна может быть совершенно незначительным, иногда почти невидимым, когда рабочая область практически исчезает. Одним из наиболее общих способов определения размера рабочей области окна является обработка сообщения WM_SIZE в оконной процедуре. Windows посылает в оконную процедуру сообщение WM_SIZE при любом изменении размеров окна. Переменная lParam, преданная в оконную процедуру, содержит ширину рабочей области в млад-

шем слове и высоту в старшем слове. Код программы для обработки этого сообщения часто выглядит следующим образом:

```
static int cxClient, cyClient;
[другие инструкции программы]
case WM_SIZE:
  cxClient = LOWORD (lParam);
  cyClient = HIWORD (lParam);
return 0;
```

Макросы LOWORD и HIWORD определяются в заголовочных файлах Windows. Такие как cxChar и cyChar, переменные cxClient и cyClient определяются внутри оконной процедуры в качестве статических, так как они используются позднее при обработке других сообщений. За сообщением WM_SIZE будет следовать WM_PAINT. Потому что при определении класса окна задан стиль класса: CS_HREDRAW | CS_VREDRAW; такой стиль класса указывает на необходимость перерисовки как при горизонтальном так и при вертикальном изменении размеров окна.

2.12 Полосы прокрутки

Полосы прокрутки предназначены для просмотра информации как в вертикальном, так и в горизонтальном направлениях. Вставить в окно приложения вертикальную или горизонтальную полосу прокрутки очень просто. Все, что нужно сделать, это включить идентификатор WS_VSCROLL (вертикальная прокрутка) и WS_HSCROLLW (горизонтальная прокрутка) или оба сразу в описание стиля окна в инструкции CreateWindow. Эти полосы прокрутки всегда размещаются у правого края или в нижней части окна и занимают всю высоту или ширину рабочей области. Рабочая область не включает в себя пространство, занятое полосами прокрутки. Ширина вертикальной полосы прокрутки и высота горизонтальной постоянны для конкретного дисплейного драйвера. Если необходимы эти значения, их можно получить, вызвав функцию GetSystemMetrics.

2.13 Диапазон и положение полос прокрутки

Каждая полоса прокрутки имеет диапазон и положение бегунка внутри диапазона. Когда бегунок находится в крайней верхней (или крайней левой) части полосы прокрутки, положение бегунка соответствует минимальному значению диапазона. Крайнее правое или крайнее нижнее положение бегунка соответствует максимальному значению диапазона. По умолчанию устанавливается диапазон полос прокрутки от 0 до 100. Но его легко изменить: SetScrollRange(hwnd, iBar, iMin, iMax, bRedraw); параметр iBar равен либо SB_VERT, либо SB_HORZ, iMin и iMax соответствуют диапазону полосы прокрутки. bRedraw устанавливается в TRUE, если не-

обходимо, чтобы Windows перерисовывала полосы прокрутки на основе вновь заданного диапазона. Для установки нового положения бегунка используется функция SetScrollPos: SetScrollPos(hwnd, iBar, iPos, bRedraw); iPos – это новое положение бегунка. Для получения текущего диапазона и положения бегунка применяют функции GetScrollRange и GetScrollPos.

2.14 Сообщения полос прокрутки

Windows посылает оконной процедуре асинхронные сообщения WM_VSCROLL и WM_HSCROLL, когда на полосе прокрутки щелкают мышью или перетаскивают бегунок. Каждое действие мыши на полосе прокрутки вызывает как минимум два сообщения, одно при нажатии кнопки, а другое при отпускании ее. Младшее слово параметра wParam, которое объединяет сообщения WM_VSCROLL и WM_HSCROLL – это число, показывающее, что мышь осуществляет какие-то действия на полосе прокрутки. Его значения соответствуют определенным идентификаторам:

Таблица 2.1 – Таблица стилей класса

Идентификатор	Описание
SB_LINEUP	Нажата клавиша мыши на верхней (левой) кнопке со стрелкой в вертикальной (горизонтальной) полосе прокрутки.
SB_ENDSCROLL	Отпущена клавиша мыши на полосе прокрутки.
SB_LINEDOWN	Нажата клавиша мыши на нижней (правой) кнопке со стрелкой в вертикальной (горизонтальной) полосе прокрутки.
SB_PAGEUP	Нажата клавиша мыши между бегунком и верхней (левой) кнопкой со стрелкой в вертикальной (горизонтальной) полосе прокрутки.
SB_PAGEDOWN	Нажата клавиша мыши между бегунком и нижней (правой) кнопкой со стрелкой в вертикальной (горизонтальной) полосе прокрутки.
SB_THUMBTRACK	Нажата клавиша на бегунке.
SB_THUMBPOSITION	Отпущена клавиша на бегунке.

Если младшее слово параметра wParam равно SB_THUMBTRACK или SB_THUMBPOSITION, то старшее слово wParam определяет текущее положение полосы прокрутки. Во всех других случаях wParam игнорируется. Также игнорируется параметр lParam, который используется в окнах диалога. Также существуют значения wParam равные SB_TOP и SB_BOTTOM. Показывающие, что полосы прокрутки переведены в свое максимальное или минимальное положение.

Обработка сообщений SB_THUMBTRACK и SB_THUMBPOSITION весьма проблематично. Если установлен большой диапазон полосы про-

крутки, а пользователь быстро перемещает бегунок по полосе, то Windows отправит вашей оконной процедуре множество сообщений SB_THUMBTRACK. Программа столкивается с проблемой обработки этих сообщений. Поэтому лучше обрабатывать сообщение SB_THUMBPOSITION, которое означает, что бегунок оставлен в покое. Однако, если есть возможность быстро обновлять содержимое экрана, обрабатывайте сообщение SB_THUMBTRACK.

2.15 Прокрутка в программе SYSMETS

Обновленный вызов функции CreateWindow добавляет вертикальную полосу прокрутки к окну, благодаря включению в описание стиля класса идентификатора WS_VSCROLL: WS_OVERLAPPEDWINDOW | WS_VSCROLL. Приводится только оконная процедура программы.

```
LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM
lParam)
     {
     static int  cxChar, cxCaps, cyChar, cyClient, iVscrollPos ;
     char        szBuffer[10] ;
     HDC         hdc ;
     int         i, y ;
     PAINTSTRUCT ps ;
     TEXTMETRIC  tm ;
     switch (iMsg)    {
          case WM_CREATE :
               hdc = GetDC (hwnd) ;
               GetTextMetrics (hdc, &tm) ;
               cxChar = tm.tmAveCharWidth ;
               cxCaps = (tm.tmPitchAndFamily & 1 ? 3 : 2)*cxChar / 2 ;
               cyChar = tm.tmHeight + tm.tmExternalLeading ;
               ReleaseDC (hwnd, hdc) ;
               SetScrollRange (hwnd, SB_VERT, 0, NUMLINES, FALSE) ;
               SetScrollPos   (hwnd, SB_VERT, iVscrollPos, TRUE) ;
               return 0 ;
          case WM_SIZE :
               cyClient = HIWORD (lParam) ;
               return 0 ;
          case WM_VSCROLL :
               switch (LOWORD (wParam))
                    {
                    case SB_LINEUP :
                         iVscrollPos -= 1 ;
                         break ;
                    case SB_LINEDOWN :
                         iVscrollPos += 1 ;
                         break ;
                    case SB_PAGEUP :
                         iVscrollPos -= cyClient / cyChar ;
                         break ;
                    case SB_PAGEDOWN :
```

```
                         iVscrollPos += cyClient / cyChar ;
                         break ;
                    case SB_THUMBPOSITION :
                         iVscrollPos = HIWORD (wParam) ;
                         break ;
                    default :
                         break ;
                    }
               iVscrollPos = max (0, min (iVscrollPos, NUMLINES)) ;
               if (iVscrollPos != GetScrollPos (hwnd, SB_VERT))
                    {
                    SetScrollPos (hwnd, SB_VERT, iVscrollPos, TRUE) ;
                    InvalidateRect (hwnd, NULL, TRUE) ;
                    }
               return 0 ;
          case WM_PAINT :
               hdc = BeginPaint (hwnd, &ps) ;

               for (i = 0 ; i < NUMLINES ; i++)
                    {
                    y = cyChar * (1 - iVscrollPos + i) ;
                    TextOut (hdc, cxChar, y, sysmetrics[i].szLabel,
strlen (sysmetrics[i].szLabel));
                    TextOut (hdc, cxChar + 22 * cxCaps, y,
sysmetrics[i].szDesc,
 strlen (sysmetrics[i].szDesc)) ;
                    SetTextAlign (hdc, TA_RIGHT | TA_TOP) ;
                    TextOut (hdc, cxChar + 22 * cxCaps + 40 * cxChar,
y, szBuffer,
                              wsprintf (szBuffer,
"%5d",GetSystemMetrics (sysmetrics[i].iIndex))) ;
                    SetTextAlign (hdc, TA_LEFT | TA_TOP) ;
                    }
               EndPaint (hwnd, &ps) ;
               return 0 ;
          case WM_DESTROY :
               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }
```

К оконной процедуре WndProc добавляются две строки для установки диапазона и положения полосы прокрутки во время обработки сообщения WM_CREATE:

```
SetScrollRange ( hwnd, SB_VERT, 0, NUMLINES, FALSE);
SetScrollPass ( hwnd, SB_VERT, iVscrollPos, TRUE);
```

Т.к. структура sysmetrics содержит NUMLINES строк, то диапазон полосы прокрутки устанавливается от 0 до NUMLINES. Каждое положение бегунка соответствует строке текста в верхней части рабочей области.

Если положение бегунка равно 0, то в окне сверху будет пустая строка, если – NUMLINES, то последняя строка.

2.16 Функция ScrollWindow

Для каждого действия с полосой прокрутки необходимо сначала рассчитать приращение ее текущей позиции при обработке сообщения WM_VSCROLL и WM_HSCROLL. Это значение затем можно использовать для прокрутки имеющегося в окне содержимого с помощью вызова функции *ScrollWindow.* Функция ScrollWindow листает содержание клиентской области определенного окна. Эта функция существует для совместимости с Win16. Для новых приложений, используют функцию ScrollWindowEx.

```
BOOL ScrollWindow(HWND hWnd, int XAmount, int YAmount,
   CONST RECT *lpRect, CONST RECT *lpClipRect);
```

hWnd – Идентифицирует окно, где клиентская область должна быть пролистана;

XAmount, YAmount – величины прокрутки в пикселя;

lpRect –Указатель на структуру RECT, определяющую блок клиентской области, которая будет пролистана. Если этот параметр NULL, вся клиентская область будет пролистана.

lpClipRect – Указатель на структуру RECT, содержащую координаты прямоугольника отсечения. Воздействуют только на биты устройства внутри прямоугольника отсечения.

Если функция выполнилась без ошибки, возвращаемое значение отлично от нуля. При сбое, возвращаемое значение - NULL. Windows делает недействительным прямоугольную зону рабочей области, открываемую операцией прокрутки. Это приводит к выдаче сообщения WM_PAIT. *InvalidateRect* больше не нужна. Функция *ScrollWindow* не является процедурой GDI, поэтому ей не нужен описатель контекста устройства.

```
int ScrollWindowEx(HWND hWnd, int dx, int dy, CONST RECT *prcScroll,
CONST RECT *prcClip, HRGN hrgnUpdate, LPRECT prcUpdate, UINT flags);
```

dx, dy – величины прокрутки в пикселях;

PrcScroll – указатель на структуру RECT, определяющую блок клиентской области, которая будет пролистана. Если этот параметр NULL, вся клиентская область пролистана.

PrcClip –указатель на структуру RECT, содержащую координаты прямоугольника отсечения.

HrgnUpdate – Идентифицирует область, которая изменяется, для объявления области прокрутки недействительной. Этот параметр может быть NULL.

PrcUpdate – указатель на структуру RECT, получающую границы недействительного прямоугольника. Этот параметр может быть NULL.

flags – Определяет флажки, которые управляют прокруткой. Этот параметр может быть одно из следующих значений:

SW_ERASE – Стирает недействительную область, посылая сообщение WM_ERASEBKGND к окну, определяется с флажком SW_INVALIDATE;

SW_INVALIDATE – Объявляет область недействительной, идентифицированную hrgnUpdate параметром после прокрутки.

SW_SCROLLCHILDREN – Листает все дочерние окна, которые пересекают прямоугольник, указанный на prcScroll параметром. Дочерние окна пролистаны числом пикселов, определенных dx и dy параметрами. Windows посылает WM_MOVE сообщение всем дочерним окнам, которые пересекают prcScroll прямоугольник.

3 Контрольные вопросы

1) При каких ситуациях возникает сообщение WM_PAINT?
2) Что такое действительные и недействительные прямоугольники?
3) Что такое контекст устройства и для чего он нужен?
4) Опишите способы получения контекста устройства.
5) Опишите функцию TextOut.
6) Что такое системный шрифт и где он применяется?
7) Для чего нужны полосы прокрутки?
8) Как вставить в рабочую область окна вертикальную и горизонтальную полосы прокрутки?
9) Какие сообщения вырабатывают полосы прокрутки?
10) Как задаётся диапазон полосы прокрутки и чему он равен изначально?

4 Задание

1) Изучить описание лабораторной работы.
2) Написать программу, которая получает контекст устройства.
3) Добавьте в рабочую область окна написанной программы текст, который превышает размеры окна по вертикали и горизонтали.
4) Отладить и протестировать полученную программу.
5) Добавьте в программу вертикальную и горизонтальную полосы прокрутки.
6) Отладить и протестировать полученную программу.
7) Написать программу в которой при передвижении бегунка текст содержащийся в окне двигался вместе с ним.
8) Отладить и протестировать полученную программу.
9) Оформить отчёт.

Лабораторная работа № 3. Использование клавиатуры в приложениях Windows

1 Цель работы

Изучить работу с клавиатурой в приложениях Windows.

2 Краткая теория

2.1 Клавиатура. Основные понятия

Основанная на сообщениях архитектура Windows идеальна для работы с клавиатурой. Ваша программа узнает о нажатиях клавиш посредством сообщений, которые посылаются оконной процедуре.

На самом деле все происходит не столь просто: когда пользователь нажимает и отпускает клавиши, драйвер клавиатуры передает информацию о нажатии клавиш в Windows. Windows сохраняет эту информацию (в виде сообщений) в системной очереди сообщений. Затем она передает сообщения клавиатуры, по одному за раз, в очередь сообщений программы, содержащей окно, имеющее "фокус ввода". Затем программа отправляет сообщения соответствующей оконной процедуре.

Смысл этого двухступенчатого процесса — сохранение сообщений в системной очереди сообщений, и дальнейшая их передача в очередь сообщений приложения — в синхронизации. Если пользователь печатает на клавиатуре быстрее, чем программа может обрабатывать поступающую информацию, Windows сохраняет информацию о дополнительных нажатиях клавиш в системной очереди сообщений, поскольку одно из этих дополнительных нажатий может быть переключением фокуса ввода на другую программу. Информацию о последующих нажатиях следует, затем направлять в другую программу. Таким образом, Windows корректно синхронизирует такие сообщения клавиатуры.

Для отражения различных событий клавиатуры, Windows посылает программам восемь различных сообщений. Это может показаться излишним, и ваша программа вполне может игнорировать многие из них.

2.2 Фокус ввода

Клавиатура должна разделяться между всеми приложениями, работающими под Windows. Некоторые приложения могут иметь больше одного окна, и клавиатура должна разделяться между этими окнами в рамках одного и того же приложения. Когда на клавиатуре нажата клавиша, только одна оконная процедура может получить сообщение об этом. Окно, которое получает это сообщение клавиатуры, является окном имеющем "фокус ввода".

Окно, имеющее фокус ввода — это либо активное окно, либо дочернее окно активного окна. Наиболее часто дочерними окнами являются

кнопки, переключатели, флажки, полосы прокрутки и списки, которые обычно присутствуют в окне диалога. Сами по себе дочерние окна никогда не могут быть активными. Если фокус ввода находится в дочернем окне, то активным является родительское окно этого дочернего окна. То, что фокус ввода находится в дочерних окнах, обычно показывается посредством мигающего курсора или каретки.

Если активное окно минимизировано, то окна с фокусом ввода нет. Windows продолжает слать программе сообщения клавиатуры, но эти сообщения выглядят иначе, чем сообщения, направленные активным и еще не минимизированным окнам.

Обрабатывая сообщения WM_SETFOCUS и WM_KILLFOCUS, оконная процедура может определить, когда окно имеет фокус ввода. WM_SETFOCUS показывает, что окно получило фокус ввода, а WM_KILLFOCUS, что окно потеряло его.

2.3 Аппаратные и символьные сообщения

Сообщения, которые приложение получает от Windows о событиях, относящихся к клавиатуре, различаются на "аппаратные" и "символьные". Такое положение соответствует двум представлениям о клавиатуре. Во-первых, вы можете считать клавиатуру набором клавиш. В клавиатуре имеется только одна клавиша <A>. Нажатие на эту клавишу является аппаратным событием. Отпускание этой клавиши являются аппаратным событием. Но клавиатура также является устройством ввода, генерирующем отображаемые символы. Клавиша <A>, в зависимости от состояния клавиш <Ctrl>, <Shift> и <CapsLock>, может стать источником нескольких символов. Обычно, этим символом является строчное 'a'. Если нажата клавиша <Shift> или установлен режим Caps Lock, то этим символом является прописное <A>. Если нажата клавиша <Ctrl>, этим символом является <Ctrl>+<A>. На клавиатуре, поддерживающей иностранные языки аппаратному событию 'A' может предшествовать либо специальная клавиша, либо <Shift>, либо <Ctrl>, либо <Alt>, либо их различные сочетания. Эти сочетания могут стать источником вывода строчного 'a' или прописного 'A' с символом ударения.

Для сочетаний аппаратных событий, которые генерируют отображаемые символы, Windows посылает программе и оба аппаратных и символьное сообщения. Некоторые клавиши не генерируют символов. Это такие клавиши, как клавиши переключения, функциональные клавиши, клавиши управления курсором и специальные клавиши, такие как Insert и Delete. Для таких клавиш Windows вырабатывает только аппаратные сообщения.

2.4 Аппаратные сообщения

Когда вы нажимаете клавишу. Windows помещает либо сообщение WM_KEYDOWN, либо сообщение WM_SYSKEYDOWN в очередь сообщений окна, имеющего фокус ввода. Когда вы отпускаете клавишу. Windows помешает либо сообщение WM_KEYUP, либо сообщение WM_SYSKEYUP в очередь сообщений.

Таблица 3.1 – Таблица сообщений клавиатуры

	Клавиша нажата	**Клавиша отпущена**
Несистемное аппаратное сообщение	WM_KEYDOWN	WM_KEYUP
Системное аппаратное сообщение	WM_SYSKEYDOWN	WM_SYSKEYUP

Обычно сообщения о "нажатии" и "отпускании" появляются парами. Однако, если вы оставите клавишу нажатой так, чтобы включился автоповтор, то Windows посылает оконной процедуре серию сообщений WM_KEYDOWN (или WM_SYSKEYDOWN) и одно сообщение WM_KEYUP (или WM_SYSKEYUP), когда в конце концов клавиша будет отпущена. Также как и все синхронные сообщения, аппаратные сообщения клавиатуры также становятся в очередь. Вы можете с помощью функции GetMessageTime получить время нажатия и отпускания клавиши относительно старта системы.

2.5 Системные и несистемные аппаратные сообщения клавиатуры

Сообщения WM_SYSKEYDOWN и WM_SYSKEYUP обычно вырабатываются при нажатии клавиш в сочетании с клавишей <Alt>. Эти сообщения вызывают опции меню программы или системного меню, или используются для системных функций, таких как смена активного окна <Alt>+<Tab> или <Alt>+<Esc>, или как быстрые клавиши системного меню (<Alt> в сочетании с функциональной клавишей). Программы обычно игнорируют сообщения WM_SYSKEYDOWN и WM_SYSKEYUP и передают их DefWindowProc. Поскольку Windows отрабатывает всю логику Alt-клавиш, то вам фактически не нужно обрабатывать эти сообщения. Ваша оконная процедура в конце концов получит другие сообщения, являющиеся результатом этих аппаратных сообщений клавиатуры (например, выбор меню). Если вы хотите включить в код вашей оконной процедуры инструкции для обработки аппаратных сообщений клавиатуры, то после обработки этих сообщений передайте их в DefWindowProc, чтобы Windows могла по-прежнему их использовать в обычных целях.

Но подумайте немного об этом. Почти все, что влияет на окно вашей программы, в первую очередь проходит через вашу оконную процедуру.

Windows каким-то образом обрабатывает сообщения только в том случае, если они передаются в DefWindowProc.

Сообщения WM_KEYDOWN и WM_KEYUP обычно вырабатываются для клавиш, которые нажимаются и отпускаются без участия клавиши <Alt>. Ваша программа может использовать или не использовать эти сообщения клавиатуры. Сама Windows их игнорирует.

2.6 Переменная lParam

Для всех аппаратных сообщений клавиатуры, 32-разрядная переменна lParam, передаваемая в оконную процедуру, состоит из шести полей: счетчик повторений, скан-кода OEM, флага расширенной клавиатуры, кода контекста, флага предыдущего состояния клавиши и флага состояния клавиши.

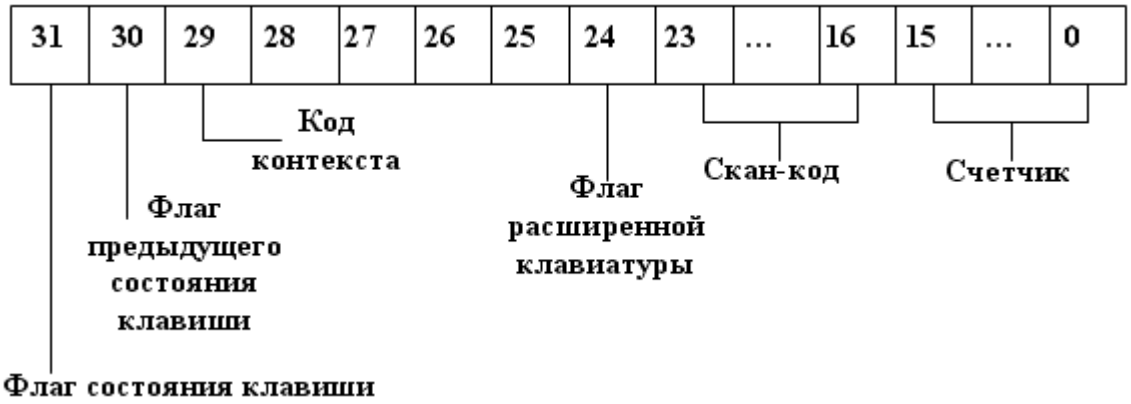

Рисунок 3.1 – Переменная lParam

2.6.1 Счетчик повторений

Счетчик повторений равен числу нажатий клавиши, которое отражено в сообщении. В большинстве случаев он устанавливается в 1. Однако, если клавиша остается нажатой, а ваша оконная процедура недостаточно быстра, чтобы обрабатывать эти сообщения в темпе автоповтора.

2.6.2 Скан-код OEM

Скан-код OEM является кодом клавиатуры, генерируемым аппаратурой компьютера.

2.6.3 Флаг расширенной клавиатуры

Флаг расширенной клавиатуры устанавливается в 1, если сообщение клавиатуры появилось в результате работы с дополнительными клавишами расширенной клавиатуры IBM. (Расширенная клавиатура IBM имеет функциональные клавиши сверху и отдельную комбинированную область клавиш управления курсором и цифр.) Этот флаг устанавливается в 1 для

клавишей <Alt> и <Ctrl> на правой стороне клавиатуры, клавиш управления курсором (включая <lnsert> и <Delete>, которые не являются частью числовой клавиатуры, клавиш наклонной черты (</> и <Enter> на числовой клавиатуре и клавиши <NumLock>. Программы для Windows обычно игнорируют флаг расширенной клавиатуры.

2.6.4 Код контекста

Код контекста устанавливается в 1, если нажата клавиша <Alt>. Этот разряд всегда равен 1 для сообщений WM_SYSKEYDOWN и WM_SYSKEYUP и 0 для сообщений WM_KEYDOWN и WM^KEYUP с двумя исключениями:

Если активное окно минимизировано, оно не имеет фокус ввода. Все нажатия клавиш вырабатывают сообщения WM_SYSKEYDOWN и WM_SYSKEYUP. Если не нажата клавиша <Alt>, поле кода контекста устанавливается в 0. (Windows использует SYS сообщения клавиатуры так, чтобы активное окно, которое минимизировано, не обрабатывало эти сообщения.)

На некоторых иноязычных клавиатурах некоторые символы генерируются комбинацией клавиш <Shift>, <Ctrl> или <Alt> с другой клавишей. В эти? случаях, у переменной lParam, которая сопровождает сообщения WM_KEYDOWN и WM_KEYUP, в поле кода контекста ставится 1, но эти сообщения не являются системными сообщениями клавиатуры.

2.6.5 Флаг предыдущего состояния клавиши

Флаг предыдущего состояния клавиши равен 0, если в предыдущем состоянии клавиша была отпущена, и 1, если в предыдущем состоянии она была нажата. Он всегда устанавливается в 1 для сообщения WM_KEYUP или WM_SYSKEYUP, но для сообщения WM_KEYDOWN или WM_SYSKEYDOWN он может устанавливаться как в 1, так и в 0. В этом случае 1 показывает наличие второго и последующих сообщений от клавиш в результате автоповтора.

2.6.6 Флаг состояния клавиши

Флаг состояния клавиши равен 0, если клавиша нажимается, и 1, если клавиша отпускается. Это поле устанавливается в 0 для сообщения WM_KEYDOWN или WM_SYSKEYDOWN и в 1 для сообщения WM_KEYUP или WM_SYSKEYUP.

2.7 Виртуальные коды клавиш

Хотя некоторая информация в lParam при обработке сообщений WM_KEYUP, WM_KEYDOWN, WM_SYSKEYUP и WM_SYSKEYDOWN может оказаться полезной, гораздо более важен параметр wParam. В этом параметре содержится "виртуальный код клавиши", идентифицирующий

нажатую или отпущенную клавишу. Разработчики Windows попытались определить виртуальные клавиши независимым от аппаратуры способом. По этой причине, некоторые виртуальные коды клавиш не могут вырабатываться на персональных компьютерах IBM и совместимых с ними, но их можно встретить на клавиатурах других производителей.

Таблица 3.2 – Виртуальные коды клавиш

Десятичное значение	**Идентификатор**	**Требуется**	**Клавиатура IBM**
1	VK_LBUTTON		
2	VK_RBUTTON		
3	VK_CANSEL	V	Ctrl-Break
4	VK_MBUTTON		
8	VK_BACK	V	Backspace
9	VK_TAB	V	Tab
12	VK_CLEAR		5 на дополнительной клавиатуре с выключенным Num lock
13	VK_RETURN	V	Enter
16	VK_SHIFT	V	Shift
17	VK_CONTROL	V	Ctrl
18	VK_MENU	V	Alt
19	VK_PAUSE		Pause
20	VK_CAPITAL	V	Caps Lock
27	VK_ESCAPE	V	Esc
32	VK_SPACE	V	Пробел
33	VK_PRIOR	V	Page up
34	VK_NEXT	V	Page down
35	VK_END	V	End
36	VK_HOME	V	Home
37	VK_LEFT	V	Стрелка влево
38	VK_UP	V	Стрелка вверх
39	VK_RIGHT	V	Стрелка вправо
40	VK_DOWN	V	Стрелка вниз
41	VK_SELECT		
42	VK_PRINT		
43	VK_EXECUTE		

44	VK_SNAPSHOT		Print screen
45	VK_INSERT	V	Insert
46	VK_DELETE	V	Delete
47	VK_HELP		
48-57		V	От 0 до 9 на основной клавиатуре
65-90		V	От A до Z
96-105	VK_NUMPAD0 - VK_NUMPAD9		От 0 до 9 на дополнительной клавиатуре при включенном Num Lock
106	VK_MULTIPLY		* на дополнительной клавиатуре
107	VK_ADD		+ на дополнительной клавиатуре
108	VK_SEPARATOR		
109	VK_SUBTRACT		- на дополнительной клавиатуре
110	VK_DECIMAL		. на дополнительной клавиатуре
111	VK_DIVIDE		/ на дополнительной клавиатуре
112-127	VK_F1 - VK_F16	V	Функциональные клавиши F1 - F16
144	VK_NUMLOCK		Num lock
145	VK_SCROLL		Scroll lock

Пометка (v) в столбце "Требуется" показывает, что клавиша предопределена для любой реализации Windows. Windows также требует, чтобы клавиатура и драйвер клавиатуры позволяли комбинировать клавиши <Shift>, <Ctrl>, а также <Shift> и <Ctrl> вместе, со всеми буквенными клавишами, всеми клавишами управления курсором и всеми функциональными клавишами. Виртуальные коды клавиш VK_LBUTTON, VK_MBUTTON и VK RBUTTON относятся к левой, центральной и правой кнопкам мыши. Однако, вам никогда не удастся получить сообщения клавиатуры с параметром wParam, в котором установлены эти значения.

2.8 Положения клавиш сдвига и клавиш-переключателей

Параметры wParam и lParam сообщений WM_KEYUP, WM_KEYDOWN, WM_SYSKEYUP и WM_SYSKEYDOWN ничего не сообщают вашей программе о положении клавиш сдвига и клавиш-переключателей. Вы можете получить текущее состояние любой виртуальной клавиши с помощью функции GetKeyState. Эта функция в основном используется для получения информации о состоянии клавиш сдвига <Shift>, <Ctrl> и <Alt> и клавиш-переключателей (<CapsLock>. <NumLock> и <ScrollLock>). Например:

```
GetKeyState (VK_SHIFT);
```

возвращает отрицательное значение (т. е., установлен старший разряд), если клавиша <Shift> нажата.

В возвращаемом функцией:

```
GetKeyState (VK_CAPITAL);
```

значении установлен младший разряд, если переключатель <CapsLock> включен. Вы также можете получить положение кнопок мыши с помощью виртуальных кодов клавиш VK_LBUTTON, VK_MBUTTON и VK_RBUTTON.

Будьте осторожны с функцией GetKeyState. Она не отражает положение клавиатуры в реальном времени. Она отражает состояние клавиатуры на момент, когда последнее сообщение от клавиатуры было выбрано из очереди. Функция GetKeyState не позволяет вам получать информацию о клавиатуре, независимо от обычных сообщений клавиатуры. Эта синхронизация, в самом деле, дает вам преимущество, потому что, если вам нужно узнать положение переключателя для конкретного сообщения клавиатуры, GetKeyState обеспечивает возможность получения точной информации, даже если вы обработаете сообщение уже после того, как состояние переключателя было изменено. Если вам действительно нужна информация о текущем положении клавиши, вы можете использовать функцию GetAsyncKeyState.

Теперь рассмотрим пример программы обрабатывающий виртуальные клавиши и связывающий их с полосами прокрутки:

```
LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM
lParam)
     {
     static int  cxChar, cxCaps, cyChar, cxClient, cyClient,
iMaxWidth,
                 iVscrollPos, iVscrollMax, iHscrollPos, iHscrollMax ;
     char        szBuffer[10] ;
     HDC         hdc ;
     int         i, x, y, iPaintBeg, iPaintEnd, iVscrollInc,
iHscrollInc ;
     PAINTSTRUCT ps ;
     TEXTMETRIC  tm ;
     switch (iMsg)
          {
          case WM_CREATE :
               hdc = GetDC (hwnd) ;
               GetTextMetrics (hdc, &tm) ;
               cxChar = tm.tmAveCharWidth ;
               cxCaps = (tm.tmPitchAndFamily & 1 ? 3 : 2) * cxChar / 2
;
               cyChar = tm.tmHeight + tm.tmExternalLeading ;
```

```
                ReleaseDC (hwnd, hdc) ;
                iMaxWidth = 40 * cxChar + 22 * cxCaps ;
                return 0 ;
           case WM_SIZE :
                cxClient = LOWORD (lParam) ;
                cyClient = HIWORD (lParam) ;
                iVscrollMax = max (0, NUMLINES + 2 - cyClient / cyChar)
;
                iVscrollPos = min (iVscrollPos, iVscrollMax) ;
                SetScrollRange (hwnd, SB_VERT, 0, iVscrollMax, FALSE) ;
                SetScrollPos   (hwnd, SB_VERT, iVscrollPos, TRUE) ;
                iHscrollMax = max (0, 2 + (iMaxWidth - cxClient) /
cxChar) ;
                iHscrollPos = min (iHscrollPos, iHscrollMax) ;
                SetScrollRange (hwnd, SB_HORZ, 0, iHscrollMax, FALSE) ;
                SetScrollPos   (hwnd, SB_HORZ, iHscrollPos, TRUE) ;
                return 0 ;
           case WM_VSCROLL :
                switch (LOWORD (wParam))
                     {
                     case SB_TOP :
                          iVscrollInc = -iVscrollPos ;
                          break ;
                     case SB_BOTTOM :
                          iVscrollInc = iVscrollMax - iVscrollPos ;
                          break ;
                     case SB_LINEUP :
                          iVscrollInc = -1 ;
                          break ;
                     case SB_LINEDOWN :
                          iVscrollInc = 1 ;
                          break ;
                     case SB_PAGEUP :
                          iVscrollInc = min (-1, -cyClient / cyChar) ;
                          break ;
                     case SB_PAGEDOWN :
                          iVscrollInc = max (1, cyClient / cyChar) ;
                          break ;
                     case SB_THUMBTRACK :
                          iVscrollInc = HIWORD (wParam) - iVscrollPos ;
                          break ;
                     default :
                          iVscrollInc = 0 ;
                     }
                iVscrollInc = max (-iVscrollPos, min (iVscrollInc,
iVscrollMax - iVscrollPos)) ;
                if (iVscrollInc != 0)
                     {
                     iVscrollPos += iVscrollInc ;
                     ScrollWindow (hwnd, 0, -cyChar * iVscrollInc,
NULL, NULL) ;
                     SetScrollPos (hwnd, SB_VERT, iVscrollPos, TRUE) ;
                     UpdateWindow (hwnd) ;
                     }
```

```
                return 0 ;
           case WM_HSCROLL :
                switch (LOWORD (wParam))
                     {
                     case SB_LINEUP :
                          iHscrollInc = -1 ;
                          break ;
                     case SB_LINEDOWN :
                          iHscrollInc = 1 ;
                          break ;
                     case SB_PAGEUP :
                          iHscrollInc = -8 ;
                          break ;
                     case SB_PAGEDOWN :
                          iHscrollInc = 8 ;
                          break ;
                     case SB_THUMBPOSITION :
                          iHscrollInc = HIWORD (wParam) - iHscrollPos ;
                          break ;
                     default :
                          iHscrollInc = 0 ;
                     }
                iHscrollInc = max (-iHscrollPos,
                              min (iHscrollInc, iHscrollMax -
iHscrollPos)) ;
                if (iHscrollInc != 0)
                     {
                     iHscrollPos += iHscrollInc ;
                     ScrollWindow (hwnd, -cxChar * iHscrollInc, 0,
NULL, NULL) ;
                     SetScrollPos (hwnd, SB_HORZ, iHscrollPos, TRUE) ;
                     }
                return 0 ;
            case WM_KEYDOWN :
                switch (wParam)
                     {
                     case VK_HOME :
                          SendMessage (hwnd, WM_VSCROLL, SB_TOP, 0L) ;
                          break ;
                     case VK_END :
                          SendMessage (hwnd, WM_VSCROLL, SB_BOTTOM, 0L)
;
                          break ;
                     case VK_PRIOR :
                          SendMessage (hwnd, WM_VSCROLL, SB_PAGEUP, 0L)
;
                          break ;
                     case VK_NEXT :
                          SendMessage (hwnd, WM_VSCROLL, SB_PAGEDOWN,
0L) ;
                          break ;
                     case VK_UP :
                          SendMessage (hwnd, WM_VSCROLL, SB_LINEUP, 0L)
;
```

```
                         break ;
                    case VK_DOWN :
                         SendMessage (hwnd, WM_VSCROLL, SB_LINEDOWN,
0L) ;
                         break ;
                    case VK_LEFT :
                         SendMessage (hwnd, WM_HSCROLL, SB_PAGEUP, 0L)
;
                         break ;
                    case VK_RIGHT :
                         SendMessage (hwnd, WM_HSCROLL, SB_PAGEDOWN,
0L) ;
                         break ;
                    }
               return 0 ;
          case WM_PAINT :
               hdc = BeginPaint (hwnd, &ps) ;
               iPaintBeg = max (0, iVscrollPos + ps.rcPaint.top /
cyChar - 1) ;
               iPaintEnd = min (NUMLINES, iVscrollPos +
ps.rcPaint.bottom / cyChar) ;
               for (i = iPaintBeg ; i < iPaintEnd ; i++)
                    {
                    x = cxChar * (1 - iHscrollPos) ;
                    y = cyChar * (1 - iVscrollPos + i) ;
                    TextOut (hdc, x, y, sysmetrics[i].szLabel, strlen
(sysmetrics[i].szLabel)) ;
                    TextOut
(hdc,x+22*cxCaps,y,sysmetrics[i].szDesc,strlen (sysmetrics[i].szDesc))
;
                    SetTextAlign (hdc, TA_RIGHT | TA_TOP) ;
                    TextOut (hdc, x + 22 * cxCaps + 40 * cxChar,
y,szBuffer,
                              wsprintf (szBuffer,
"%5d",GetSystemMetrics (sysmetrics[i].iIndex))) ;
                    SetTextAlign (hdc, TA_LEFT | TA_TOP) ;
                    }
               EndPaint (hwnd, &ps) ;
               return 0 ;
          case WM_DESTROY :
               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }
```

2.9 Символьные сообщения

Вы уже встречали такой код раньше:

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

```
}
```

Это типичный цикл обработки сообщений, содержащийся в WinMain. Функция GetMessage заполняет поля структуры msg данными следующего сообщения из очереди. Вызов DispatchMessage вызывает соответствующую оконную процедуру.

Между двумя этими функциями находится функция TranslateMessage, преобразующая аппаратные сообщения клавиатуры в символьные сообщения. Если этим сообщением является WM_KEYDOWN или WM_SYSKEYDOWN и, если нажатие клавиши в сочетании с положением клавиши сдвига генерирует символ, тогда TranslateMessage помещает символьное сообщение в очередь сообщений. Это символьное сообщение будет следующим, после сообщения о нажатии клавиши, которое функция GetMessage извлечет из очереди сообщений.

Существует четыре символьных сообщения:

Таблица 3.3 – Виртуальные коды клавиш

	Символы	**Немые символы**
Не системные символы	WM_CHAR	WM_DEADCHAR
Системные символы	WM_SYSCHAR	WM_SYSDEADCHAR

Сообщения WM_CHAR и WM_DEADCHAR являются следствием сообщений WM_KEYDOWN. Сообщения WM_SYSCHAR и WM_SYSDEADCHAR являются следствием сообщений WM_SYSKEYDOWN. В большинстве случаев ваши программы для Windows могут игнорировать все сообщения, за исключением WM_CHAR. Параметр lParam, передаваемый в оконную процедуру как часть символьного сообщения, является таким же, как параметр lParam аппаратного сообщения клавиатуры, из которого сгенерировано символьное сообщение. Параметр wParam – это код символа ASCII.

2.10 Сообщения WM_CHAR

Если вашей Windows-программе необходимо обрабатывать символы клавиатуры (например, в программах обработки текстов или коммуникационных программах), то она будет обрабатывать сообщения WM_CHAR. Вероятно, вы захотите как-то по особому обрабатывать клавиши <Backspace>, <Tab> и <Enter>, но все остальные символы вы будете обрабатывать похожим образом:

```
case WM_CHAR:
switch (wParam)
{
case '/b':  // Backspace
[другие строки программы]
```

```
break;
case '/t':  // Tab
[другие строки программы]
break;
case '/n':  // перевод строки
[другие строки программы]
break;
case '/r':  // Enter
[другие строки программы]
break;
default:
[другие строки программы]
break;
}
return 0;
```

Этот фрагмент программы фактически идентичен обработке символов клавиатуры в обычных программах MS_DOS.

2.11 Сообщения немых символов

Программы для Windows обычно могут игнорировать сообщения WM_DEADCHAR и WM_SYSDEADCHAR. На некоторых, не американских клавиатурах, некоторые клавиши определяются добавлением диакритического знака к букве. Они называются "немыми клавишами", поскольку эти клавиши сами по себе не определяют символов.

Если пользователь нажимает немую клавишу, оконная процедура получает сообщение WM_DEADCHAR с параметром wParam равным коду ASCII самого диакритического знака. Когда затем пользователь нажимает клавишу буквы, оконная процедура получает сообщение WM_CHAR, где параметр wParam равен коду ASCII буквы с диакритическим знаком. Таким образом, ваша программа не должна обрабатывать сообщение WM_DEADCHAR, поскольку сообщение WM_CHAR и так дает программе всю необходимую информацию. Windows имеет даже встроенную систему отслеживания ошибок: если за немой клавишей следует буква, у которой не может быть диакритического знака (например буква s), то оконная процедура получает два сообщения WM_CHAR подряд – первое с wParam равным коду ASCII самого диакритического знака (такое же значение wParam, как было передано с сообщением WM_DEADCHAR) и второе wParam равным коду ASCII буквы s.

2.12 Вывод сообщений от клавиатуры

Эта программа выводит в рабочую область всю информацию, которую Windows посылает оконной процедуре для восьми различных сообщений клавиатуры.

```
#include <windows.h>
#include <stdio.h>
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
RECT rect ;
int  cxChar, cyChar ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "KeyLook" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;
     wndclass.cbSize        = sizeof (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 ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;
     RegisterClassEx (&wndclass) ;
     hwnd = CreateWindow (szAppName, "Keyboard Message Looker",
                          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 ;
     }
void ShowKey (HWND hwnd, int iType, char *szMessage,
              WPARAM wParam, LPARAM lParam)
     {
     static char *szFormat[2] = { "%-14s %3d    %c %6u %4d %3s %3s %4s
%4s",
                                  "%-14s    %3d %c %6u %4d %3s %3s %4s
%4s" } ;
     char        szBuffer[80] ;
     HDC         hdc ;
     ScrollWindow (hwnd, 0, -cyChar, &rect, &rect) ;
     hdc = GetDC (hwnd) ;
     SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;
     TextOut (hdc, cxChar, rect.bottom - cyChar, szBuffer,
              wsprintf (szBuffer, szFormat [iType],
                        szMessage, wParam,
```

```
                         (BYTE) (iType ? wParam : ' '),
                         LOWORD (lParam),
                         HIWORD (lParam) & 0xFF,
                         (PSTR) (0x01000000 & lParam ? "Yes"  : "No"),
                         (PSTR) (0x20000000 & lParam ? "Yes"  : "No"),
                         (PSTR) (0x40000000 & lParam ? "Down" : "Up"),
                         (PSTR) (0x80000000 & lParam ? "Up"   :
"Down"))) ;
     ReleaseDC (hwnd, hdc) ;
     ValidateRect (hwnd, NULL) ;
     }
LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM
lParam)
     {
     static char szTop[] =
               "Message        Key Char Repeat Scan Ext ALT Prev
Tran";
     static char szUnd[] =
               "-------        --- ---- ------ ---- --- --- ----
____";
     HDC         hdc ;
     PAINTSTRUCT ps ;
     TEXTMETRIC  tm ;
     switch (iMsg)
          {
          case WM_CREATE :
               hdc = GetDC (hwnd) ;
               SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT))
;
               GetTextMetrics (hdc, &tm) ;
               cxChar = tm.tmAveCharWidth ;
               cyChar = tm.tmHeight ;
               ReleaseDC (hwnd, hdc) ;
               rect.top = 3 * cyChar / 2 ;
               return 0 ;
          case WM_SIZE :
               rect.right  = LOWORD (lParam) ;
               rect.bottom = HIWORD (lParam) ;
               UpdateWindow (hwnd) ;
               return 0 ;
          case WM_PAINT :
               InvalidateRect (hwnd, NULL, TRUE) ;
               hdc = BeginPaint (hwnd, &ps) ;
               SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT))
;
               SetBkMode (hdc, TRANSPARENT) ;
               TextOut (hdc, cxChar, cyChar / 2, szTop, (sizeof szTop)
- 1) ;
               TextOut (hdc, cxChar, cyChar / 2, szUnd, (sizeof szUnd)
- 1) ;
               EndPaint (hwnd, &ps) ;
               return 0 ;
          case WM_KEYDOWN :
               ShowKey (hwnd, 0, "WM_KEYDOWN", wParam, lParam) ;
```

```
            return 0 ;
        case WM_KEYUP :
            ShowKey (hwnd, 0, "WM_KEYUP", wParam, lParam) ;
            return 0 ;
        case WM_CHAR :
            ShowKey (hwnd, 1, "WM_CHAR", wParam, lParam) ;
            return 0 ;
        case WM_DEADCHAR :
            ShowKey (hwnd, 1, "WM_DEADCHAR", wParam, lParam) ;
            return 0 ;
        case WM_SYSKEYDOWN :
            ShowKey (hwnd, 0, "WM_SYSKEYDOWN", wParam, lParam) ;
            break ;        // ie, call DefWindowProc
        case WM_SYSKEYUP :
            ShowKey (hwnd, 0, "WM_SYSKEYUP", wParam, lParam) ;
            break ;        // ie, call DefWindowProc
        case WM_SYSCHAR :
            ShowKey (hwnd, 1, "WM_SYSCHAR", wParam, lParam) ;
            break ;        // ie, call DefWindowProc
        case WM_SYSDEADCHAR :
            ShowKey (hwnd, 1, "WM_SYSDEADCHAR", wParam, lParam) ;
            break ;        // ie, call DefWindowProc
        case WM_DESTROY :
            PostQuitMessage (0) ;
            return 0 ;
        }
    return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
    }
```

Когда программа получает аппаратные сообщения клавиатуры, вызывается функция ScrollWindow для прокрутки содержимого всей рабочей области окна так, чтобы это содержимое сместилось вверх на высоту одного символа. Функция TextOut используется для вывода строки новой информации на экран, начиная с высоты одного символа от нижнего края рабочей области. В первом столбце показаны сообщения клавиатуры, во втором — коды виртуальных клавиш для аппаратных сообщений клавиатуры, в третьем — коды символов (и сами символы) для символьных сообщений и, наконец, в шести оставшихся столбцах показано состояние шести полей параметра сообщения IParam.

2.13 Функции работы с кареткой

Здесь перечислены пять основных функций работы с кареткой:
- CreateCaret — создает связанную с окном каретку.
- SetCaretPos — устанавливает положение каретки в окне.
- ShowCaret — показывает каретку.
- Hide Caret — прячет каретку.
- DestroyCaret — удаляет каретку.

Кроме этих, еще имеется функция получения положения каретки (GetCaretPos) и функции установки и получения частоты мигания каретки (GetCaretBlinkTime) и (SetCaretBlinkTime).

Каретка – это обычно горизонтальная черточка или прямоугольник, имеющие размер символа, или вертикальная черточка. Вертикальная черточка рекомендуется при использовании пропорционального шрифта, такого как задаваемый по умолчанию системный шрифт Windows. Поскольку размер символов пропорционального шрифта не фиксирован, горизонтальной черточке и прямоугольнику невозможно задать размер символа.

Вы не можете просто создать каретку при обработке сообщения WM_CREATE и удалить ее при обработке сообщения WM_DESTROY. Каретка — это то, что называется "общесистемным ресурсом". Это означает, что в системе имеется только одна каретка. И, как результат, программа при необходимости вывода каретки на экран своего окна "заимствует" ее у системы.

Серьезно ли это необычное ограничение? Конечно нет. Подумайте: появяе-ние каретки в окне имеет смысл только в том случае, если окно имеет фокус ввода. Каретка показывает пользователю, что он может вводить в программу текст. В каждый конкретный момент времени только одно окно имеет фокус ввода, поэтому для всей системы и нужна только одна каретка.

Обрабатывая сообщения WM_SETFOCUS и WM_KILLFOCUS, программа может определить, имеет ли она фокус ввода. Оконная процедура получает сообщение WM_SETFOCUS, когда получает фокус ввода, а сообщение WM_KILLFOCUS, когда теряет фокус ввода. Эти сообщения приходят попарно: оконная процедура получает сообщение WM_SETFOCUS всегда до того, как получит сообщение WM_KILLFOCUS, и она всегда получает одинаковое количество сообщений WM_SETFOCUS и WM_KILLFOCUS за время существования окна.

Основное правило использования каретки выглядит просто: оконная процедура вызывает функцию CreateCaret при обработке сообщения WM_SETFOCUS и функцию DestroyCaret при обработке сообщения WM_KILLFOCUS.

Имеется несколько других правил: каретка создается скрытой. После вызова функции CreateCaret, оконная процедура должна вызвать функцию ShowCaret, чтобы сделать каретку видимой. В дополнение к этому, оконная процедура, когда она рисует в своем окне при обработке сообщения, отличного от WM_PAINT, должна скрыть каретку, вызвав функцию HideCaret. После того как оконная процедура закончит рисовать в своем окне, она вызывает функцию ShowCaret, чтобы снова вывести каретку на экран. Функция HideCaret имеет дополнительный эффект: если вы несколько раз вызываете HideCaret, не вызывая при этом ShowCaret, то чтобы каретка

снова стала видимой, вам придется такое же количество раз вызвать функцию ShowCaret.

2.14 Программа TYPER

Вы можете набрать в окне текст, перемещая курсор (имеется ввиду каретка) по экрану с помощью клавиш управления курсором (или клавишами управления кареткой?), и стереть содержимое окна, нажав <Esc>. Окно также очистится при изменении размеров окна.

Для простоты в программе TYPER используется фиксированный шрифт. Создание текстового редактора для пропорционального шрифта является как вы могли бы сообразить, гораздо более трудным делом. Программа получает контекст устройства в нескольких местах: при обработке сообщений WM_CREATE, WM_KEYDOWN, WM_CHAR и WM_PAINT. Каждый раз при этом для выбора фиксированного шрифта вызываются функции GetStockObject и SelectObject.

При обработке сообщения WM_SETFOCUS программа TYPER вызывает функцию CreateCaret для создания каретки, имеющей ширину и высоту символа, функцию SetCaretPos для установки положения каретки и функцию ShowCaret, чтобы сделать каретку видимой. При обработке сообщения WM_KILLFOCUS программа вызывает функции HideCaret и DestroyCaret.

Программа Typer.c

```
#include <windows.h>
#include <stdlib.h>
#define BUFFER(x,y) *(pBuffer + y * cxBuffer + x)
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
     {
     static char szAppName[] = "Typer" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASSEX  wndclass ;
     wndclass.cbSize        = sizeof (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 ;
     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ;
     RegisterClassEx (&wndclass) ;
     hwnd = CreateWindow (szAppName, "Typing Program",
```

```
                         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 iMsg, WPARAM wParam, LPARAM
lParam)
     {
     static char *pBuffer = NULL ;
     static int  cxChar, cyChar, cxClient, cyClient, cxBuffer,
cyBuffer, xCaret, yCaret ;
     HDC         hdc ;
     int         x, y, i ;
     PAINTSTRUCT ps ;
     TEXTMETRIC  tm ;
switch (iMsg)
          {
          case WM_CREATE :
               hdc = GetDC (hwnd) ;
SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT)) ;
               GetTextMetrics (hdc, &tm) ;
               cxChar = tm.tmAveCharWidth ;
               cyChar = tm.tmHeight ;
ReleaseDC (hwnd, hdc) ;
               return 0 ;
case WM_SIZE :
cxClient = LOWORD (lParam) ;
cyClient = HIWORD (lParam) ;
cxBuffer = max (1, cxClient / cxChar) ;
cyBuffer = max (1, cyClient / cyChar) ;
if (pBuffer != NULL) free (pBuffer) ;
               if ((pBuffer = (char *) malloc (cxBuffer * cyBuffer))
== NULL)
                    MessageBox (hwnd, "Window too large.  Cannot "
                                      "allocate enough memory.",
"Typer",
                                      MB_ICONEXCLAMATION | MB_OK) ;
               else
                    for (y = 0 ; y < cyBuffer ; y++)
                         for (x = 0 ; x < cxBuffer ; x++)
                              BUFFER(x,y) = ' ' ;
xCaret = 0 ;
               yCaret = 0 ;
if (hwnd == GetFocus ())
                    SetCaretPos (xCaret * cxChar, yCaret * cyChar) ;
return 0 ;
```

```
          case WM_SETFOCUS :
CreateCaret (hwnd, NULL, cxChar, cyChar) ;
               SetCaretPos (xCaret * cxChar, yCaret * cyChar) ;
               ShowCaret (hwnd) ;
               return 0 ;
case WM_KILLFOCUS :
HideCaret (hwnd) ;
               DestroyCaret () ;
               return 0 ;
          case WM_KEYDOWN :
               switch (wParam)
                    {
                    case VK_HOME :
                         xCaret = 0 ;
                         break ;
                    case VK_END :
                         xCaret = cxBuffer - 1 ;
                         break ;
                    case VK_PRIOR :
                         yCaret = 0 ;
                         break ;
                    case VK_NEXT :
                         yCaret = cyBuffer - 1 ;
                         break ;
                    case VK_LEFT :
                         xCaret = max (xCaret - 1, 0) ;
                         break ;
                    case VK_RIGHT :
                         xCaret = min (xCaret + 1, cxBuffer - 1) ;
                         break ;
                    case VK_UP :
                         yCaret = max (yCaret - 1, 0) ;
                         break ;
                    case VK_DOWN :
                         yCaret = min (yCaret + 1, cyBuffer - 1) ;
                         break ;
                    case VK_DELETE :
                         for (x = xCaret ; x < cxBuffer - 1 ; x++)
                         BUFFER (x, yCaret) = BUFFER (x + 1, yCaret) ;
                         BUFFER (cxBuffer - 1, yCaret) = ' ' ;
                         HideCaret (hwnd) ;
                         hdc = GetDC (hwnd) ;
                         SelectObject (hdc,
                              GetStockObject (SYSTEM_FIXED_FONT)) ;
                         TextOut (hdc, xCaret * cxChar, yCaret *
cyChar,
                                  & BUFFER (xCaret, yCaret),
                                  cxBuffer - xCaret) ;
                         ShowCaret (hwnd) ;
                         ReleaseDC (hwnd, hdc) ;
                         break ;
                    }
               SetCaretPos (xCaret * cxChar, yCaret * cyChar) ;
```

```
                return 0 ;
           case WM_CHAR :
                for (i = 0 ; i < (int) LOWORD (lParam) ; i++)
                     {
                     switch (wParam)
                          {
                          case '\b' :                    // backspace
                               if (xCaret > 0)
                                    {
                                    xCaret-- ;
                                    SendMessage (hwnd, WM_KEYDOWN,
                                                 VK_DELETE, 1L) ;
                                    }
                               break ;

                          case '\t' :                    // tab
                               do
                                    {
                                    SendMessage (hwnd, WM_CHAR, ' ',
1L) ;
                                    }
                               while (xCaret % 8 != 0) ;
                               break ;
                          case '\n' :                    // line feed
                               if (++yCaret == cyBuffer)
                                    yCaret = 0 ;
                               break ;
                          case '\r' :                    // carriage
return
                               xCaret = 0 ;
                               if (++yCaret == cyBuffer)
                                    yCaret = 0 ;
                               break ;
                          case '\x1B' :                  // escape
                               for (y = 0 ; y < cyBuffer ; y++)
                                    for (x = 0 ; x < cxBuffer ; x++)
                                         BUFFER (x, y) = ' ' ;
                               xCaret = 0 ;
                               yCaret = 0 ;
                               InvalidateRect (hwnd, NULL, FALSE) ;
                               break ;
                          default :                       // character
codes
                               BUFFER (xCaret, yCaret) = (char) wParam

;
                               HideCaret (hwnd) ;
                               hdc = GetDC (hwnd) ;
                               SelectObject (hdc,
                                    GetStockObject (SYSTEM_FIXED_FONT))
;
                               TextOut (hdc, xCaret * cxChar, yCaret *
cyChar,
                                        & BUFFER (xCaret, yCaret), 1) ;
                               ShowCaret (hwnd) ;
```

```
                              ReleaseDC (hwnd, hdc) ;
                              if (++xCaret == cxBuffer)
                                   {
                                   xCaret = 0 ;
                                   if (++yCaret == cyBuffer)
                                        yCaret = 0 ;
                                   }
                              break ;
                         }
                    }
               SetCaretPos (xCaret * cxChar, yCaret * cyChar) ;
               return 0 ;
          case WM_PAINT :
               hdc = BeginPaint (hwnd, &ps) ;
               SelectObject (hdc, GetStockObject (SYSTEM_FIXED_FONT))
;
               for (y = 0 ; y < cyBuffer ; y++)
                    TextOut (hdc, 0, y * cyChar, & BUFFER(0,y),
cxBuffer) ;
               EndPaint (hwnd, &ps) ;
               return 0 ;
          case WM_DESTROY :
               PostQuitMessage (0) ;
               return 0 ;
          }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }
```

Сообщения WM_KEYDOWN и WM_KEYUP здесь обрабатываются более полно. Обработка сообщения WM_KEYDOWN в основном включает в себя обработку клавиш управления курсором. Клавиши <Home> и <End> заставляют каретку переместиться в начало или конец строки, клавиши <PageUp> и <PageDown> — к верхней или нижней границе окна. Клавиши стрелок работают так, как вы и ожидаете. При нажатии клавиши <Delele> программа TYPER должна переместить все, что находится в буфере, начиная от следующей позиции каретки и до конца строки, а затем вывести на экран в конец строки символ пробела.

Обработчик сообщения WM_CHAR включает в себя обработку клавиш <Backspace>, <Tab>, <Linefeed> <Ctrl>+<Enter>, <Enter>, <Escape> и символьных клавиш. Отметьте, что поле повторения параметра lParam использовано при обработке сообщения WM_CHAR (здесь предполагается, что важен каждый вводимый пользователем символ), а не при обработке сообщения WM_KEYDOWN (чтобы предотвратить нечаянное двойное нажатие). Обработка нажатий <Backspace> и <Tab> отчасти упрощена путем использования функции SendMessage. Логика обработки клавиши <Backspace> заменяется логикой обработки <Delete>, а клавише <Tab> ставится в соответствие несколько пробелов.

Как уже упоминалось, во время рисования в окне при обработке отличного от WM_PAINT сообщения, вы должны сделать каретку невидимой. В программе это делается при обработке сообщения WM_KEYDOWN для клавиши <Delete> и сообщения WM_CHAR для символьных клавиш. В обоих этих случаях, в программе TYPER меняется содержимое буфера, а затем в окне рисуется новый символ или символы.

3 Контрольные вопросы

1) Объясните понятие фокус ввода.
2) Чем отличаются системные и несистемные аппаратные сообщения клавиатуры.
3) Что такое виртуальный код клавиши и каково его применение.
4) Какую роль играет переменная IParam.
5) Для чего существуют сообщения немых символов.
6) Назавите основные функций работы с кареткой.

4 Задание

1) Изучить описание лабораторной работы.
2) Составить приложение Windows, максимально используя возможности работы с клавиатурой, предоставляемые операционной системой.
3) Отладить программу.
4) Протестировать полученную программу.
5) Оформить отчёт.

Лабораторная работа № 4. Программирование манипулятора типа "мышь"

1 Цель работы

Научится работать с мышью, используя функции WinAPI.

2 Краткая теория

2.1 Базовые знания о мыши

Windows поддерживает однокнопочную, двухкнопочную или трехкнопочную мышь, а также позволяет использовать джойстик или световое перо для имитации однокнопочной мыши. Вы можете определить наличие мыши с помощью функции *GetSystemMetrics:*

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

Значение *fMouse* будет равным TRUE (ненулевым), если мышь установлена. Для определения количества кнопок установленной мыши используйте следующий вызов:

```
cButtons = GetSystemMetrics (SM_CMOUSEBOTTONS);
```

Если мышь не инсталлирована, то возвращаемым значением этой функции будет 0.

Когда пользователь Windows перемещает мышь, Windows перемещает по экрану маленькую растровую картинку, которая называется "курсор мыши". Курсор мыши имеет "вершину" размером в один пиксель, точно указывающее положение мыши на экране.

В драйвере дисплея содержатся несколько ранее определенных курсоров мыши, которые могут использоваться в программах. Наиболее типичным курсором является наклонная стрелка, которая называется IDC_ARROW и определяется в заголовочных файлах Windows. Вершина – это конец стрелки. Курсор IDC_CROSS (используемый в приведенных в этой главе программах BLOKOUT) имеет вершину в центре крестообразного шаблона. Курсор IDC_WAIT в виде песочных часов обычно используется программами для индикации того, что они чем-то заняты. Программисты также могут спроектировать свои собственные курсоры. Курсор, устанавливаемый по умолчанию, для конкретного окна задается при определении структуры класса окна. Например:

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

Ниже приведены определения терминов, соответствующих вашим действиям над кнопками мыши:

- Щелчок – нажатие и отпускание кнопки мыши
- Двойной щелчок – двойное быстрое одно за другим нажатие и отпускание кнопки мыши
- Перетаскивание – перемещение мыши при нажатой кнопке

На трехкнопочной мыши кнопки называются левой кнопкой, средней кнопкой и правой кнопкой. В связанных с мышью идентификаторах, определенных в заголовочных файлах Windows, используются аббревиатуры LBUTTON, MBUTTON и RBUTTON. Двухкнопочная мышь имеет только левую и правую кнопки. Единственная кнопка однокнопочной мыши является левой.

2.2 Сообщения мыши, связанные с рабочей областью окна

Оконная процедура получает сообщения мыши и когда мышь проходит через окно и при щелчке внутри окна, даже если окно неактивно или не имеет фокуса ввода. В Windows для мыши определен набор из 21 сообщения. Однако 11 из этих сообщений не относятся к рабочей области, и программы для Windows обычно игнорируют их.

Если мышь перемещается по рабочей области окна, оконная процедур" получает сообщение WM_MOUSEMOVE. Если кнопка мыши нажи-

мается или отпускается внутри рабочей области окна, оконная процедура получает следующие сообщения:

Таблица 4.1 – Виртуальные коды клавиш

Кнопка	**Нажатие**	**Отпускание**	**Нажатие (Второй щелчок)**
Левая	WM_LBUTTONDOWN	WM_LBUTTONUP	WM_LBUTTONDBLCLK
Средняя	WM_MBUTTONDOWN	WM_MBUTTONUP	WM_MBUTTONDBLCLK
Правая	WM_RBUTTONDOWN	WM_RBUTTONUP	WM_RBUTTONDBLCLK

Ваша оконная процедура получает сообщения "MBUTTON" только при наличии трехкнопочной мыши и сообщения "RBUTTON" только при наличии двух- или трехкнопочной мыши. Оконная процедура получает сообщения "DBLCLK" (двойной щелчок) только в том случае, если класс окна был определен так, чтобы их можно было получать.

Для всех этих сообщений значение параметра *lParam* содержит положение мыши. Младшее слово — это координата *x*, а старшее слово — координата *y* относительно верхнего левого угла рабочей области окна. Вы можете извлечь координаты *x* и *y* из параметра *lParam* с помощью макросов LOWORD и HIWORD, определенных в заголовочных файлах Windows. Значение параметра *wParam* показывает состояние кнопок мыши и клавиш <Shift> и <Ctrl>. Вы можете проверить параметр *wParam* с помощью битовых масок, определенных в заголовочных файлах. Префикс MK означает "клавиша мыши".

```
MK_LBUTTON          Левая кнопка нажата
MK_MBUTTON          Средняя кнопка нажата
MK_RBUTTON          Правая кнопка нажата
MK_SHIFT            Клавиша <Shift> нажата
MK_CONTROL          Клавиша <Ctrl> нажата
```

При движении мыши по рабочей области окна, Windows не вырабатывает сообщение WM_MOUSEMOVE для всех возможных положений мыши. Количество сообщений WM_MOUSEMOVE, которые получает ваша программа, зависит от устройства мыши и от скорости, с которой ваша оконная процедура может обрабатывать сообщения о движении мыши.

Если вы щелкните левой кнопкой мыши в рабочей области неактивного окна. Windows сделает активным окно, в котором вы произвели щелчок, и затем передаст оконной процедуре сообщение WM_LBUTTONDOWN. Если ваша оконная процедура получает сообщение WM_LBUTTONDOWN, то ваша программа может уверенно считать, что ее окно активно. Однако, ваша оконная процедура может получить сообщение WM_LBUTTONUP, не получив вначале сообщения

WM_LBUTTONDOWN. Это может случиться, если кнопка мыши нажимается в одном окне, мышь перемещается в ваше окно, и кнопка отпускается. Аналогично, оконная процедура может получить сообщение WM_LBUTTONDOWN без соответствующего ему сообщения WM_LBUTTONUP, если кнопка мыши отпускается во время нахождения в другом окне.

Из этих правил есть два исключения:

- Оконная процедура может "захватить мышь" и продолжать получать сообщения мыши, даже если она находится вне рабочей области окна.
- Если системное модальное окно сообщений или системное модальное окно диалога находится на экране, никакая другая программа не может получать сообщения мыши. Системные модальные окна сообщений и диалога запрещают переключение на другое окно программы, пока оно активно. (Примером системного модального окна сообщений является окно, которое появляется, когда вы завершаете работу с Windows.)

2.3 Пример обработки сообщений мыши

```
//CONNECT.C
/* connect.c - win & mouse */
#include <windows.h>
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
#define MAXPOINTS 1000
#define MoveTo(hdc,x,y) MoveToEx(hdc,x,y,NULL);

#pragma argsused
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static char szAppName[] = "Connect" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASS  wndclass ;
/*     wndclass.cbSize        = sizeof (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 ;
/*     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ; */
     RegisterClass (&wndclass) ;
     hwnd = CreateWindow (szAppName, "Connect-the-Points Mouse Demo",
```

```
                         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 iMsg, WPARAM wParam, LPARAM
lParam)
{
     static POINT points[MAXPOINTS] ;
     static int   iCount ;
     HDC          hdc ;
     PAINTSTRUCT  ps ;
     int          i, j ;
     switch (iMsg)
          {
          case WM_LBUTTONDOWN :
               iCount = 0 ;
               InvalidateRect (hwnd, NULL, TRUE) ;
               return 0 ;
          case WM_MOUSEMOVE :
               if (wParam & MK_LBUTTON && iCount < 1000)
                    {
                    points[iCount  ].x = LOWORD (lParam) ;
                    points[iCount++].y = HIWORD (lParam) ;
                    hdc = GetDC (hwnd) ;
                    SetPixel (hdc, LOWORD (lParam), HIWORD (lParam),
0L) ;
                    ReleaseDC (hwnd, hdc) ;
                    }
               return 0 ;
          case WM_LBUTTONUP :
               InvalidateRect (hwnd, NULL, FALSE) ;
               return 0 ;
          case WM_PAINT :
               hdc = BeginPaint (hwnd, &ps) ;
               SetCursor (LoadCursor (NULL, IDC_WAIT)) ;
               ShowCursor (TRUE) ;
               for (i = 0 ; i < iCount - 1 ; i++)
                    for (j = i + 1 ; j < iCount ; j++)
                         {
                         MoveTo (hdc, points[i].x, points[i].y) ;
                         LineTo (hdc, points[j].x, points[j].y) ;
                         }
               ShowCursor (FALSE) ;
               SetCursor (LoadCursor (NULL, IDC_ARROW)) ;
```

```
            EndPaint (hwnd, &ps) ;
            return 0 ;
        case WM_DESTROY :
            PostQuitMessage (0) ;
            return 0 ;
        }
    return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
    }
```

Данная программа обрабатывает три сообщения мыши:

- WM_LBUTTONDOWN — Программа очищает рабочую область.
- WM_MOUSEMOVE — Если левая кнопка мыши нажата, то программа рисует черную точку в рабочей области в текущем положении мыши.
- WM_LBUTTONUP — Программа соединяет все точки, нарисованные в рабочей области, друг с другом.

Поскольку программе для рисования линий может потребоваться некоторое время, при обработке сообщения WM_PAINT она изменяет вид курсора на песочные часы, а после окончания рисования, возвращает курсор в предыдущее состояние. Для этого требуется два вызова функции *SetCursor,* в которых используются два стандартных курсора. В программе также дважды вызывается функция *ShowCursor,* первый раз с параметром TRUE и второй — с параметром FALSE.

Если программа занята рисованием линий, вы можете нажать кнопку мыши, подвигать мышью и отпустить кнопку мыши, но ничего не произойдет. Программа не получает эти сообщения, поскольку она занята и не может сделать ни одного вызова *GetMessage.* После того, как программа закончит рисование линий, она опять не получает этих сообщений, поскольку кнопка мыши к этому времени отпущена. В этом отношении мышь не похожа на клавиатуру. Windows обращается с каждой нажатой на клавиатуре клавишей как с чем-то важным. Однако, если кнопка мыши нажата и отпущена в рабочей области, пока программа занята, щелчки мыши просто сбрасываются.

2.3.1 Обработка клавиш <Shift>

Когда описанная выше программа получает сообщение WM_MOUSEMOVE, она выполняет поразрядную операцию AND со значениями *wParam* и MK_LBUTTON для определения того, нажата ли левая кнопка. Вы также можете использовать *wParam* для определения состояния клавиш <Shift>. Например, если обработка должна зависеть от состояния клавиш <Shift> и <Ctrl>, то вы могли бы воспользоваться следующей логикой:

```
if (MK_SHIFT & wParam)
```

```
if (MK_CONTROL & wParam)
{
[нажаты клавиши <Shift> и <Ctrl>]
}
else
{
[нажата клавиша <Shift>]
}
else if(MK_CONTROL & wParam)
{
[нажата клавиша <Ctrl>]
}
else
{
[клавиши <Shift> и <Ctrl> не нажаты]
}
```

Если вы хотите в вашей программе использовать и левую и правую кнопки мыши, и если вы также хотите обеспечить возможность работы пользователям однокнопочной мыши, вы можете так написать вашу программу, чтобы действие клавиши <Shift> в сочетании с левой кнопкой мыши было тождественно действию правой кнопки. В этом случае ваша обработка щелчков кнопки могла бы выглядеть так:

```
case WM_LBUTTONDOWN:
if(!MK_SHIFT & wParam)
{
[логика обработки левой кнопки]
return 0;
}
case WM_RBUTTONDOWN:
[логика обработки правой кнопки]
return 0;
```

Функция *GetKeyState* также может возвращать состояние кнопок мыши или клавиш <Shift>, используя виртуальные коды клавиш VK_LBUTTON, VK_RBUTTON, VK_MBUTTON, VK_SHIFT и VK_CONTROL. При нажатой кнопке или клавише возвращаемое значение функции *GetKeyState* отрицательно. Функция *GetKeyState* возвращает состояние мыши или клавиши в связи с обрабатываемым в данный момент сообщением, т. е. информация о состоянии должным образом синхронизируется с сообщениями. Но поскольку вы не можете использовать функцию *GetKeyState* для клавиши, которая еще только должна быть нажата, ее нельзя использовать и для кнопки мыши, которая еще только должна быть нажата. Не делайте так:

```
while (GetKeyState (VK_LBUTTON) >= 0); // ОШИБКА!!!
```

функция *GetKeyState* сообщит о том, что левая кнопка нажата только в том случае, если левая кнопка уже нажата, когда вы обрабатываете сообщение и вызываете *GetKeyState.*

2.3.2 Двойные щелчки клавиш мыши

Двойным щелчком мыши называются два, следующих один за другим в быстром темпе, щелчка мыши. Для того, чтобы два последовательных щелчка мыши считались двойным щелчком, они должны произойти в течение очень короткого промежутка времени, который называется "временем двойного щелчка". Если вы хотите, чтобы ваша оконная процедура получала сообщения двойного щелчка мыши, то вы должны включить идентификатор CS_DBLCLKS при задании стиля окна в классе окна перед вызовом функции *RegisterClassEx:*

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

Если вы не включите CS_DBLCLKS в стиль окна, и пользователь дважды в быстром темпе щелкнет левой кнопкой мыши, то ваша оконная процедура получит следующие сообщения: WM_LBUTTONDOWN, WM_LBUTTONUP, WM_LBUTTONDOWN и WM_LBUTTONUP. (Оконная процедура вполне может между этими сообщениями от кнопок мыши получать и другие сообщения.) Если вы хотите реализовать собственную логику обработки двойного щелчка мыши, то для получения относительного времени сообщений WM_LBUTTONDOWN, вы можете использовать функцию Windows *GetMessageTime.*

Если вы включаете в свой класс окна идентификатор CS_DBLCLKS, то оконная процедура при двойном щелчке мыши получает следующие сообщения: WM_LBUTTONDOWN, WM_LBUTTONUP, WM_LBUTTONDBLCLK и WM_LBUTTONUP. Сообщение WM_LBUTTONDBLCLK просто заменяет второе сообщение WM_LBUTTONDOWN.

Двойной щелчок мыши гораздо легче обрабатывать, если первый щелчок выполняет в оконной процедуре те же самые действия, которые выполняет в оконной процедуре те же самые действия, которые выполняет простой щелчок. Затем второй щелчок (сообщение WM_LBUTTONBLCLK) выполняет какие-то дополнительные, относительно первого щелчка, действия.

2.3.3 Сообщения мыши нерабочей области

Те десять сообщений мыши, которые мы только что обсудили, относятся к ситуации, когда действия с мышью, т. е. перемещения мыши и щелчки ее кнопками, происходят в рабочей области окна. Если мышь ока-

зывается вне рабочей области окна, но все еще внутри окна. Windows посылает оконной процедуре сообщения мыши "нерабочей области". Нерабочая область включает в себя панель заголовка, меню и полосы прокрутки окна.

Обычно вам нет необходимости обрабатывать сообщения мыши нерабочей области. Вместо этого вы просто передаете их в *DefWindowProc,* чтобы Windows могла выполнить системные функции. В этом смысле сообщения мыши нерабочей области похожи на системные сообщения клавиатуры WM_SYSKEYDOWN, WM_SYSKEYUP и WM_SYSCHAR.

Сообщения мыши нерабочей области почти полностью такие же как и сообщения мыши рабочей области. В названия сообщений входят буквы "NC", что означает "нерабочая" (nonclient). Если мышь перемещается внутри нерабочей области окна, то оконная процедура получает сообщение WM_NCMOUSEMOVE. Нажатие кнопки мыши вырабатывают следующие сообщения:

Таблица 4.2 – Виртуальные коды клавиш

Кнопка	**Нажатие**	**Отпускание**	**Нажатие (Второй щелчок)**
Левая	WM_NCLBUTTONDOWN	WM_NCLBUTTONUP	WM_NCLBUTTONDBLCLK
Средняя	WM_NCMBUTTONDOWN	WM_NCMBUTTONUP	WM_NCMBUTTONDBLCLK
Правая	WM_NCRBUTTONDOWN	WM_NCRBUTTONUP	WM_NCRBUTTONDBLCLK

Однако, параметры *wParam* и *lParam* для сообщений мыши нерабочей области отличаются от соответствующих параметров для сообщений мыши рабочей области. Параметр *wParam* показывает зону нерабочей области, в которой произошло перемещение или щелчок мыши. Его значение приравнивается одному из идентификаторов, начинающихся с HT (что означает "тест попадания"), которые определяется в заголовочных файлах Windows.

Переменная *lParam* содержит в младшем слове значение координаты *x,* а в старшем — *y.* Однако, эти координаты являются координатами экрана, а не координатами рабочей области, как это было у сообщений мыши рабочей области. Значения координат *x* и *y* верхнего левого угла экрана равны 0. Если вы движетесь вправо, то увеличивается значение координаты *x,* если вниз, то значение координаты *y.*

Вы можете преобразовать экранные координаты в координаты рабочей области окна и наоборот с помощью двух функций Windows:

```
ScreenToClient(hwnd, pPoint);
ClientToScreen(hwnd, pPoint);
```

Параметр *pPoint* — это указатель на структуру тира POINT. Две эти функции преобразуют хранящиеся в структуре данные без сохранения их прежних значений. Отметьте, что если точка с экранными координатами находится выше рабочей области окна, то преобразованное значение координаты *y* рабочей области будет отрицательным. И аналогично, значение *x* экранной координаты левее рабочей области после преобразования в координату рабочей области станет отрицательным.

2.3.4Сообщение теста попадания

Если вы вели подсчет, то знаете, что мы рассмотрели 20 из 21 сообщения мыши. Последним сообщением является WM_NCHITTEST, означающее "тест попадания в нерабочую область". Это сообщение предшествует всем остальным сообщениям мыши рабочей и нерабочей области. Параметр *lParam* содержит значения *x* и *y* экранных координат положения мыши. Параметр *wParam* не используется.

В приложениях для Windows это сообщение обычно передается в DefWindowProc. В этом случае Windows использует сообщение WM_NCHITTEST для выработки всех остальных сообщений мыши на основе положения мыши. Для сообщений мыши нерабочей области возвращаемое значение функции *DefWindowProc* при обработке сообщения WM_NCHITTEST передается как параметр *wParam* в сообщении мыши. Это значение может быть любым из множества значений *wParam,* которое бывает у этого параметра для сообщений мыши нерабочей области, плюс следующие:

```
HTCLIENT          Рабочая область
HTNOWHERE         Нет ни на одном из окон
HTTRANSPARENT     Окно перекрыто другим окном
HTERROR      Заставляет DefWindowProc генерировать гудок
```

Если функция *DefWindowProc* после обработки сообщения WM_NCHITTEST возвращает значение HTCLIENT, то Windows преобразует экранные координаты в координаты рабочей области вырабатывает сообщение мыши рабочей области.

Если вы вспомните, как мы запретили все системные функции клавиатуры при обработке сообщения WM_SYSKEYDOWN, то вы, наверное, удивитесь, если сможете сделать что-нибудь подобное, используя сообщения мыши. Действительно, если вы вставите строки:

```
case WM_NCHITTEST:
return (LRESULT) HTNOWHERE;
```

в вашу оконную процедуру, то вы полностью запретите все сообщения мыши рабочей и нерабочей области вашего окна. Кнопки мыши про-

сто не будут работать до тех пор, пока мышь будет находится где-либо внутри вашего окна, включая значок системного меню, кнопки минимизации, максимизации и закрытия окна.

2.4 Сообщения порождают сообщения

Windows использует сообщение WM_NCHITTEST для выработки всех остальных сообщений мыши. Идея сообщений, порождающих другие сообщения, характерна для Windows. Давайте рассмотрим пример. Если вы дважды щелкните мышью на значке системного меню Windows-программы, то программа завершится. Двойной щелчок генерирует серию сообщений WM_NCHITTEST. Поскольку мышь установлена над значком системного меню, то возвращаемым значением функции *DefWindowProc* является HTSYSMENU, и Windows ставит в очередь сообщение WM_NCLBUTTONDBLCLK с параметром *wParam,* равным HTSYSMENU.

Оконная процедура обычно передает это сообщение *DefWindowProc.* Когда функция *DefWindowProc* получает сообщение WM_NCLBUTTONDBLCLK с параметром *wParam,* равным HTSYSMENU, то она ставит в очередь сообщение WM_SYSCOMMAND с параметром *wParam,* равным SC_CLOSE. (Это сообщение WM_SYSCOMMAND также генерируется, когда пользователь выбирает в системном меню пункт Close.) Оконная процедура вновь передает это сообщение в *DefWindowProc. DefWindowProc* обрабатывает сообщение, отправляя оконной процедуре синхронное сообщение WM_CLOSE.

Если программе для своего завершения требуется получить от пользователя подтверждение, оконная процедура может обработать сообщение WM_CLOSE. В противном случае оконная процедура обрабатывает сообщение WM_CLOSE, вызывая функцию *DestroyWindow.* Помимо других действий, функция *DestroyWindow* посылает оконной процедуре синхронное сообщение WM_DESTROY. Обычно оконная процедура обрабатывает сообщение WM_DESTROY следующим образом:

```
case WM_DESTROY:
PostQuitMessage(0) ;
return 0;
```

Функция *PostQuitMessage* заставляет Windows поместить в очередь сообщений сообщение WM_QUIT. До оконной процедуры это сообщение никогда не доходит, поскольку оно является причиной того, что функция *GetMessage* возвращает 0, что вызывает завершение цикла обработки сообщений и программы в целом.

2.5 Тестирование попадания

```
/* checker1.c  */
#include <windows.h>
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
#define DIVISIONS 5
#define MoveTo(hdc,x,y) MoveToEx(hdc,x,y,NULL);

#pragma argsused
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static char szAppName[] = "Checker1" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASS  wndclass ;
/*     wndclass.cbSize        = sizeof (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 ;
/*     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ; */
  RegisterClass (&wndclass) ;
  hwnd = CreateWindow (szAppName, "Checker1 Mouse Hit-Test Demo",
                           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 iMsg, WPARAM wParam, LPARAM
lParam)
{
     static BOOL fState[DIVISIONS][DIVISIONS] ;
     static int  cxBlock, cyBlock ;
     HDC         hdc ;
     PAINTSTRUCT ps ;
     RECT        rect ;
     int         x, y ;
```

```
  switch (iMsg)
  {
    case WM_SIZE :
      cxBlock = LOWORD (lParam) / DIVISIONS ;
      cyBlock = HIWORD (lParam) / DIVISIONS ;
      return 0 ;

    case WM_LBUTTONDOWN :
      x = LOWORD (lParam) / cxBlock ;
      y = HIWORD (lParam) / cyBlock ;
      if (x < DIVISIONS && y < DIVISIONS)
      {
        fState [x][y] ^= 1 ;
        rect.left   = x * cxBlock ;
        rect.top    = y * cyBlock ;
        rect.right  = (x + 1) * cxBlock ;
        rect.bottom = (y + 1) * cyBlock ;
        InvalidateRect (hwnd, &rect, FALSE) ;
      }
      else  MessageBeep (0) ;
      return 0 ;

    case WM_PAINT :
      hdc = BeginPaint (hwnd, &ps) ;
      for (x = 0 ; x < DIVISIONS ; x++)
        for (y = 0 ; y < DIVISIONS ; y++)
        {
          Rectangle (hdc, x * cxBlock, y * cyBlock,
                                  (x + 1) * cxBlock, (y + 1) * cyBlock)
;
          if (fState [x][y])
          {
            MoveTo (hdc,  x    * cxBlock,  y    * cyBlock) ;
            LineTo (hdc, (x+1) * cxBlock, (y+1) * cyBlock) ;
            MoveTo (hdc,  x    * cxBlock, (y+1) * cyBlock) ;
            LineTo (hdc, (x+1) * cxBlock,  y    * cyBlock) ;
          }
        }
      EndPaint (hwnd, &ps) ;
      return 0 ;

    case WM_DESTROY :
      PostQuitMessage (0) ;
      return 0 ;
  }
  return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}
```

В программе у всех 25 прямоугольников одинаковая высота и ширина. Значения ширины и высоты хранятся в *cxBlock* и *cyBlock,* и пересчитываются при изменении размеров рабочей области. В логике обработки сообщений WM_LBUTTONDOWN для определения прямоугольника, на ко-

тором был произведен щелчок, используются координаты мыши. Этот обработчик устанавливает текущее состояние прямоугольника в массиве *fState,* и делает соответствующий прямоугольник недействительным для выработки сообщения WM_PAINT. Если ширина или высота рабочей области не делится без остатка на пять, узкая полоса справа или внизу рабочей области не будет зарисована прямоугольниками. В ответ на щелчок мыши в этой области, программа, вызывая функцию *MessageBeep,* сообщит об ошибке.

Когда программа получает сообщение WM_PAINT, она перерисовывает всю рабочую область, рисуя прямоугольники с помощью функции GDI *Rectangle.* Если установлено значение *fState,* то программа с помощью функций *MoveTo* и *LineTo* рисует две линии. Перед перерисовкой, при обработке сообщения WM_PAINT, программа не проверяет, действительна ли каждая прямоугольная область, хотя могла бы это делать. Первый метод такой проверки заключается в создании внутри цикла структуры RECT для каждого прямоугольного блока (с использованием тех же формул, что и для сообщения WM_LBUTTONDOWN) и проверки, с помощью функции *IntersectRect,* пересекается ли он с недействительным прямоугольником *(ps.rcPaint).* Другим методом может быть использование функции *PtInRect* для определения того, находится ли любой из четырех углов прямоугольного блока внутри недействительного прямоугольника.

2.6 Эмуляция мыши с помощью клавиатуры

Даже если мышь не установлена. Windows все же может вывести курсор мыши на экран. Windows поддерживает для курсора "счетчик отображения". Если мышь инсталлирована, то начальное значение счетчика отображения равно 0; если нет, то его начальное значение равно -1. Курсор мыши выводится на экран только в том случае, если значение счетчика отображения неотрицательно. Вы можете увеличить значение счетчика отображения на 1, вызывая функцию:

```
ShowCursor (TRUE);
```

и уменьшить его на 1 с помощью вызова:

```
ShowCursor (FALSE);
```

Перед использованием функции *ShowCursor* нет нужды определять, инсталлирована мышь или нет. Если вы хотите вывести на экран курсор мыши независимо от наличия собственно мыши, просто увеличьте на 1 счетчик отображения. После однократного увеличения счетчика отображения, его уменьшение скроет курсор только в том случае, если мышь не была инсталлирована, но при наличии мыши курсор останется на экране.

Вывод на экран курсора мыши относится ко всей операционной системе Windows, поэтому вам следует быть уверенным, что вы увеличивали и уменьшали счетчик отображения равное число раз.

В вашей оконной процедуре можете использовать следующую простую логику:

```
case WM_SETFOCUS: ShowCursor (TRUE);
return 0;
case WM_KILLFOCUS: ShowCursor (FALSE) ;
return 0;
```

Оконная процедура получает сообщение WM_SETFOCUS, когда окно приобретает фокус ввода клавиатуры, и сообщение WM_KILLFOCUS, когда теряет фокус ввода. Эти события являются самыми подходящими для того, чтобы вывести на экран или убрать с экрана курсор мыши. Во-первых, сообщения WM_SETFOCUS и WM_KILLFOCUS являются взаимодополняющими, — следовательно, оконная процедура равное число раз увеличит и уменьшит счетчик отображения курсора. Во-вторых, для версий Windows, в которых мышь не инсталлирована, использование сообщений WM_SETFOCUS и WM_KILLFOCUS вызовет появление курсора только в том окне, которое имеет фокус ввода. И только в этом случае, используя интерфейс клавиатуры, который вы создадите, пользователь сможет перемещать курсор.

Windows отслеживает текущее положение курсора мыши, даже если сама мышь не инсталлирована. Если мышь не установлена, и вы выводите курсор мыши на экран, то он может оказаться в любой части экрана и будет оставаться там до тех пор, пока вы не переместите его. Получить положение курсора можно с помощью функции:

```
GetCursorPos (pPoint);
```

где *pPoint* — это указатель на структуру POINT. Функция заполняет поля структуры POINT значениями координат *x* и *y* мыши. Установить положение курсора можно с помощью функции:

```
SetCursorPos (x, y) ;
```

В обоих этих случаях значения координат *x* и *y* являются координатами экрана, а не рабочей области. (Это очевидно, поскольку этим функциям не нужен параметр *hwnd.)* Как уже отмечалось, вы можете преобразовать экранные координаты в координаты рабочей области и наоборот с помощью функций *ScreenToClient* и ClientToScreen.

Если вы при обработке сообщения мыши вызываете функцию *GetCursorPos* и преобразуете экранные координаты в координаты рабочей об-

ласти, то они могут слегка отличаться от координат, содержащихся в параметре *lParam* сообщения мыши. Координаты, которые возвращаются из функции *GetCursorPos,* показывают текущее положение мыши. Координаты в параметре *lParam* сообщения мыши — это координаты мыши в момент генерации сообщения.

Вы, вероятно, захотите так написать логику работы клавиатуры, чтобы двигать курсор мыши с помощью клавиш управления курсором клавиатуры и воспроизводить кнопки мыши с помощью клавиши пробела <Spacebar> или <Enter>. При этом не следует делать так, чтобы курсор мыши сдвигался на один пиксель за одно нажатие клавиши. В этом случае вам для того. чтобы переместить указатель мыши от одной стороны экрана до другой, надо было бы держать клавишу со стрелкой нажатой более минуты.

Если нужно реализовать интерфейс клавиатуры так, чтобы точность позиционирования курсора мыши по-прежнему оставалась равной одному пикселю, тогда обработка сообщения клавиатуры должна идти следующим образом: при нажатии клавиши со стрелкой курсор мыши начинает двигаться медленно, а затем, если клавиша остается нажатой, его скорость возрастает. Вспомните, что параметр *lParam* в сообщениях WM_KEYDOWN показывает, являются ли сообщения о нажатии клавиш результатом автоповтора. Превосходное применение этой информации!

Рассмотрим пример с использованием клавиатуры:

```
/* checker2.c  */
#include <windows.h>
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
#define DIVISIONS 5
#define MoveTo(hdc,x,y) MoveToEx(hdc,x,y,NULL);

#pragma argsused
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static char szAppName[] = "BlokOut1" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASS  wndclass ;
/*     wndclass.cbSize        = sizeof (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 ;
/*     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ; */
     RegisterClass (&wndclass) ;
     hwnd = CreateWindow (szAppName, "Checker2 Mouse Hit-Test Demo",
                          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 iMsg, WPARAM wParam, LPARAM
lParam)
{
     static BOOL fState[DIVISIONS][DIVISIONS] ;
     static int  cxBlock, cyBlock ;
     HDC         hdc ;
     PAINTSTRUCT ps ;
     POINT       point ;
     RECT        rect ;
     int         x, y ;

  switch (iMsg)
  {
    case WM_SIZE :
      cxBlock = LOWORD (lParam) / DIVISIONS ;
      cyBlock = HIWORD (lParam) / DIVISIONS ;
      return 0 ;

    case WM_SETFOCUS :
      ShowCursor (TRUE) ;
      return 0 ;

    case WM_KILLFOCUS :
      ShowCursor (FALSE) ;
      return 0 ;

    case WM_KEYDOWN :
      GetCursorPos (&point) ;
      ScreenToClient (hwnd, &point) ;
      x = max (0, min (DIVISIONS - 1, point.x / cxBlock)) ;
      y = max (0, min (DIVISIONS - 1, point.y / cyBlock)) ;
      switch (wParam)
      {
        case VK_UP :
                          y-- ;
                          break;
```

```
        case VK_DOWN :
                        y++ ;
                        break ;
        case VK_LEFT :
                        x-- ;
                        break ;
        case VK_RIGHT :
                        x++ ;
                        break ;
        case VK_HOME :
                        x = y = 0 ;
                        break ;
        case VK_END :
                        x = y = DIVISIONS - 1 ;
                        break ;
        case VK_RETURN :
        case VK_SPACE :
                        SendMessage (hwnd, WM_LBUTTONDOWN,
MK_LBUTTON,
                                MAKELONG (x * cxBlock, y *
cyBlock)) ;
                        break ;
      }
      x = (x + DIVISIONS) % DIVISIONS ;
      y = (y + DIVISIONS) % DIVISIONS ;
      point.x = x * cxBlock + cxBlock / 2 ;
      point.y = y * cyBlock + cyBlock / 2 ;
      ClientToScreen (hwnd, &point) ;
      SetCursorPos (point.x, point.y) ;
      return 0 ;

    case WM_LBUTTONDOWN :
      x = LOWORD (lParam) / cxBlock ;
      y = HIWORD (lParam) / cyBlock ;
      if (x < DIVISIONS && y < DIVISIONS)
      {
        fState[x][y] ^= 1 ;
        rect.left   = x * cxBlock ;
        rect.top    = y * cyBlock ;
        rect.right  = (x + 1) * cxBlock ;
        rect.bottom = (y + 1) * cyBlock ;
        InvalidateRect (hwnd, &rect, FALSE) ;
      }
      else MessageBeep (0);
      return 0 ;

    case WM_PAINT :
      hdc = BeginPaint (hwnd, &ps) ;
      for (x = 0 ; x < DIVISIONS ; x++)
        for (y = 0 ; y < DIVISIONS ; y++)
        {
          Rectangle (hdc, x * cxBlock, y * cyBlock,
                          (x + 1) * cxBlock, (y + 1) * cyBlock) ;
          if (fState [x][y])
```

```
            {
              MoveTo (hdc,  x    * cxBlock,  y    * cyBlock) ;
              LineTo (hdc, (x+1) * cxBlock, (y+1) * cyBlock) ;
              MoveTo (hdc,  x    * cxBlock, (y+1) * cyBlock) ;
              LineTo (hdc, (x+1) * cxBlock,  y    * cyBlock) ;
            }
          }
        EndPaint (hwnd, &ps) ;
        return 0 ;

      case WM_DESTROY :
        PostQuitMessage (0) ;
        return 0 ;
    }
    return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}
```

Логика обработки сообщения WM_KEYDOWN в программе следующая: определяется положение курсора *(GetCursorPos),* координаты экрана преобразуются в координаты рабочей области *(ScreenToClient),* а затем координаты делятся на ширину и высоту прямоугольного блока. Полученные значения *x* и *y* показывают положение прямоугольника в массиве 5 на 5. Курсор мыши при нажатии клавиши не всегда находится в рабочей области, поэтому *x* и *y* должны быть обработаны макросами *min* и *max.* таким образом, чтобы гарантировать их попадание в диапазон от 0 до 4.

Для клавиш со стрелками программа увеличивает или уменьшает на 1 соответственно значение *x* или *y.* Если нажатой клавишей является клавиша <Enter> (VK_RETURN) или клавиша <Spacebar> (VK_SPACE), то программа использует функцию SendMessage для посылки себе же синхронного сообщения WM_LBUTTONDOWN.

2.7 Захват мыши

Оконная процедура обычно получает сообщения мыши только тогда, когда курсор мыши находится в рабочей или в нерабочей области окна. Но программе может понадобиться получать сообщения мыши и тогда, когда курсор мыши находится вне окна. Если это так, то программа может произвести "захват" мыши.

Для того, чтобы понять, для чего может понадобиться захват мыши, давайте рассмотрим программу

Пример 4.2 Рисование прямоугольника

```
/* blokout1.c - draw rectangle */
#include <windows.h>
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
#pragma argsused
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
```

```
     static char szAppName[] = "BlokOut1" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASS  wndclass ;
/*     wndclass.cbSize        = sizeof (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 ;
/*     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ; */
     RegisterClass (&wndclass) ;
     hwnd = CreateWindow (szAppName, "Mouse Button Demo",
                          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 ;
}

void DrawBoxOutline (HWND hwnd, POINT ptBeg, POINT ptEnd)
{
     HDC hdc ;
     hdc = GetDC (hwnd) ;
     SetROP2 (hdc, R2_NOT) ;
     SelectObject (hdc, GetStockObject (NULL_BRUSH)) ;
     Rectangle (hdc, ptBeg.x, ptBeg.y, ptEnd.x, ptEnd.y) ;
     ReleaseDC (hwnd, hdc) ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM
lParam)
{
     static BOOL  fBlocking, fValidBox ;
     static POINT ptBeg, ptEnd, ptBoxBeg, ptBoxEnd ;
     HDC          hdc ;
     PAINTSTRUCT  ps ;
     switch (iMsg)
          {
          case WM_LBUTTONDOWN :
               ptBeg.x = ptEnd.x = LOWORD (lParam) ;
               ptBeg.y = ptEnd.y = HIWORD (lParam) ;
```

```
                DrawBoxOutline (hwnd, ptBeg, ptEnd) ;
                SetCursor (LoadCursor (NULL, IDC_CROSS)) ;
                fBlocking = TRUE ;
                return 0 ;
           case WM_MOUSEMOVE :
                if (fBlocking)
                     {
                     SetCursor (LoadCursor (NULL, IDC_CROSS)) ;
                     DrawBoxOutline (hwnd, ptBeg, ptEnd) ;
                     ptEnd.x = LOWORD (lParam) ;
                     ptEnd.y = HIWORD (lParam) ;
                     DrawBoxOutline (hwnd, ptBeg, ptEnd) ;
                     }
                return 0 ;
           case WM_LBUTTONUP :
                if (fBlocking)
                     {
                     DrawBoxOutline (hwnd, ptBeg, ptEnd) ;
                     ptBoxBeg   = ptBeg ;
                     ptBoxEnd.x = LOWORD (lParam) ;
                     ptBoxEnd.y = HIWORD (lParam) ;
                     SetCursor (LoadCursor (NULL, IDC_ARROW)) ;
                     fBlocking = FALSE ;
                     fValidBox  = TRUE ;
                     InvalidateRect (hwnd, NULL, TRUE) ;
                     }
                return 0 ;
           case WM_CHAR :
                if (fBlocking & wParam == '\x1B')       // ie, Escape
                     {
                     DrawBoxOutline (hwnd, ptBeg, ptEnd) ;
                     SetCursor (LoadCursor (NULL, IDC_ARROW)) ;
                     fBlocking = FALSE ;
                     }
                return 0 ;
           case WM_PAINT :
                hdc = BeginPaint (hwnd, &ps) ;
                if (fValidBox)
                     {
                     SelectObject (hdc, GetStockObject (BLACK_BRUSH)) ;
                     Rectangle (hdc, ptBoxBeg.x, ptBoxBeg.y,
                                     ptBoxEnd.x, ptBoxEnd.y) ;
                     }
                if (fBlocking)
                     {
                     SetROP2 (hdc, R2_NOT) ;
                     SelectObject (hdc, GetStockObject (NULL_BRUSH)) ;
                     Rectangle (hdc, ptBeg.x, ptBeg.y, ptEnd.x,
ptEnd.y) ;
                     }
                EndPaint (hwnd, &ps) ;
                return 0 ;
           case WM_DESTROY :
                PostQuitMessage (0) ;
```

```
                return 0 ;
    }
    return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}
```

Вы начинаете рисовать, нажимая левую кнопку мыши, чтобы указать один из углов прямоугольника, и удерживая кнопку нажатой, вы двигаете мышь. Программа рисует контур прямоугольника, со вторым противоположным углом в текущей позиции курсора. После отпускания кнопки мыши, программа закрашивает прямоугольник. При нажатии левой кнопки мыши, программа сохраняет координаты мыши и первый раз вызывает функцию DrawBoxOutline. Функция рисует прямоугольник с использованием растровой операции R2_NOT, которая меняет цвет рабочей области на противоположный. При обработке последующих сообщений WM_MOUSEMOVE программа снова рисует такой же прямоугольник, полностью стирая предыдущий. Затем она использует новые координаты мыши для рисования нового прямоугольника. Наконец, когда программа получает сообщение WM_LBUTTONUP, координаты мыши сохраняются, и окно делается недействительным, генерируя сообщение WM_PAINT для вывода на экран полученного прямоугольника.

Попытайтесь сделать следующее: нажмите левую кнопку мыши внутри рабочей области окна программы BLOKOUTI, а затем переместите курсор за пределы окна. Программа перестает получать сообщения WM_MOUSEMOVE. Теперь отпустите кнопку. Программа не получит сообщение WM_LBUTTONUP, поскольку курсор находится вне рабочей области. Верните курсор внутрь рабочей области окна программы. Оконная процедура по-прежнему считает, что кнопка остается нажатой. Это нехорошо. Программа не знает что происходит.

Для решения проблем такого типа был введен механизм захвата мыши. Если пользователь двигает мышь, то выход мыши за границы окна не должен создавать трудностей. Программа должна по-прежнему контролировать мышь.

Захватить мышь проще, чем поймать ее в мышеловку. Вам достаточно только вызвать функцию:

```
SetCapture (hwnd);
```

После вызова этой функции. Windows посылает все сообщения мыши в оконную процедуру того окна, описателем которого является *hwnd.* Сообщения мыши всегда остаются сообщениями рабочей области, даже если мышь оказывается в нерабочей области окна. Параметр *lParam* по-прежнему показывает положение мыши в координатах рабочей области.

Эти координаты, однако, могут стать отрицательными, если мышь окажется левее или выше рабочей области.

Пока мышь захвачена, системные функции клавиатуры тоже не действуют. Когда вы захотите освободить мышь, вызовите функцию:

```
ReleaseCapture ();
```

Эта функция возвращает обработку мыши в нормальный режим.

Рассмотрим пример захвата мыши.

```
/* blokout2.c */
#include <windows.h>
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

#pragma argsused
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static char szAppName[] = "BlokOut2" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASS  wndclass ;
/*     wndclass.cbSize        = sizeof (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 ;
/*     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ; */
     RegisterClass (&wndclass) ;
     hwnd = CreateWindow (szAppName, "Mouse Button & Capture Demo",
                          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 ;
}

void DrawBoxOutline (HWND hwnd, POINT ptBeg, POINT ptEnd)
{
```

```
     HDC hdc ;
     hdc = GetDC (hwnd) ;
     SetROP2 (hdc, R2_NOT) ;
     SelectObject (hdc, GetStockObject (NULL_BRUSH)) ;
     Rectangle (hdc, ptBeg.x, ptBeg.y, ptEnd.x, ptEnd.y) ;
     ReleaseDC (hwnd, hdc) ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM
lParam)
{
     static BOOL  fBlocking, fValidBox ;
     static POINT ptBeg, ptEnd, ptBoxBeg, ptBoxEnd ;
     HDC          hdc ;
     PAINTSTRUCT  ps ;
     switch (iMsg)
          {
          case WM_LBUTTONDOWN :
               ptBeg.x = ptEnd.x = LOWORD (lParam) ;
               ptBeg.y = ptEnd.y = HIWORD (lParam) ;
               DrawBoxOutline (hwnd, ptBeg, ptEnd) ;
               SetCapture (hwnd) ;
               SetCursor (LoadCursor (NULL, IDC_CROSS)) ;
               fBlocking = TRUE ;
               return 0 ;
          case WM_MOUSEMOVE :
               if (fBlocking)
                    {
                    SetCursor (LoadCursor (NULL, IDC_CROSS)) ;
                    DrawBoxOutline (hwnd, ptBeg, ptEnd) ;
                    ptEnd.x = LOWORD (lParam) ;
                    ptEnd.y = HIWORD (lParam) ;
                    DrawBoxOutline (hwnd, ptBeg, ptEnd) ;
                    }
               return 0 ;
          case WM_LBUTTONUP :
               if (fBlocking)
                    {
                    DrawBoxOutline (hwnd, ptBeg, ptEnd) ;
                    ptBoxBeg   = ptBeg ;
                    ptBoxEnd.x = LOWORD (lParam) ;
                    ptBoxEnd.y = HIWORD (lParam) ;
                    ReleaseCapture () ;
                    SetCursor (LoadCursor (NULL, IDC_ARROW)) ;
                    fBlocking = FALSE ;
                    fValidBox  = TRUE ;
                    InvalidateRect (hwnd, NULL, TRUE) ;
                    }
               return 0 ;
          case WM_CHAR :
               if (fBlocking & wParam == '\x1B')       // i.e., Escape
                    {
                    DrawBoxOutline (hwnd, ptBeg, ptEnd) ;
                    ReleaseCapture () ;
```

```
                    SetCursor (LoadCursor (NULL, IDC_ARROW)) ;
                    fBlocking = FALSE ;
                    }
               return 0 ;
          case WM_PAINT :
               hdc = BeginPaint (hwnd, &ps) ;
               if (fValidBox)
                    {
                    SelectObject (hdc, GetStockObject (BLACK_BRUSH)) ;
                    Rectangle (hdc, ptBoxBeg.x, ptBoxBeg.y,
                                    ptBoxEnd.x, ptBoxEnd.y) ;
                    }
               if (fBlocking)
                    {
                    SetROP2 (hdc, R2_NOT) ;
                    SelectObject (hdc, GetStockObject (NULL_BRUSH)) ;
                    Rectangle (hdc, ptBeg.x, ptBeg.y, ptEnd.x,
ptEnd.y) ;
                    }
               EndPaint (hwnd, &ps) ;
               return 0 ;
          case WM_DESTROY :
               PostQuitMessage (0) ;
               return 0 ;
     }
     return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}
```

Эта программа аналогична предыдущей, за исключением трех новых строчек кода: вызова функции *SetCapture* при обработке сообщения WM_LBUTTONDOWN и вызовов функции *ReleaseCapture* при обработке сообщений WM_LBUTTONUP и WM_CHAR. (Обработка сообщения WM_CHAR позволяет отказаться от захвата мыши при нажатии пользователем клавиши <Escape>.)

Проверьте работу программы теперь: измените размер окна так, чтобы оно стало меньше экрана целиком, начните рисовать прямоугольник внутри рабочей области, а затем уведите курсор за границы рабочей области влево или вниз, и наконец, отпустите кнопку мыши. Программа получит координаты целого прямоугольника. Чтобы его увидеть, увеличьте окно.

Вам следует всегда пользоваться захватом мыши, когда нужно отслеживать сообщения WM_MOUSEMOVE после того, как кнопка мыши была нажата в вашей рабочей области, и до момента отпускания кнопки. Ваша программа станет проще, и ожидания пользователя будут удовлетворены.

3 Контрольные вопросы

1) Каки сообщения сыши вы знаете?
2) Какую роль играет переменная lParam.

3) Какую роль играет переменная wParam.
4) Как скрыть курсор мыши.
5) Как программно переместить курсор мыши .
6) Объясните понятие «захват мыши».

4 Задание

1) Изучить описание лабораторной работы.
2) Составить приложение Windows, максимально используя возможности работы с мышью, предоставляемые операционной системой.
3) Отладить и протестировать полученную программу.
4) Оформить отчёт.

Лабораторная работа № 5. Использование системного таймера в приложениях Windows

1 Цель работы

Научиться использовать системный таймер.

2 Краткая теория

Таймер в Windows является устройством ввода информации, которое периодически извещает приложение о том, что истек заданный интервал времени. При этом Windows посылает вашей программе периодические сообщения WM_TIMER, сигнализируя об истечении интервала времени.

Сначала таймер Windows может показаться менее важным устройством ввода, чем клавиатура или мышь, и, конечно, это верно для многих приложений Windows. Но таймер более полезен, чем вы можете подумать, и не только для программ, которые индицируют время. Существуют также и другие применения таймера в Windows, некоторые из которых, может быть не столь очевидны:

- Многозадачность — хотя Windows является вытесняющей многозадачной средой, иногда самое эффективное решение для программы — как можно быстрее вернуть управление Windows. Если программа должна выполнять большой объем работы, она может разделить задачу на части и отрабатывать каждую часть при получении сообщения WM_TIMER.
- Поддержка обновления информации о состоянии — программа может использовать таймер для вывода на экран обновляемой в "реальном времени" (real-time), постоянно меняющейся информации, связанной либо с системными ресурсами, либо с процессом выполнения определенной задачи.

- Реализация "автосохранения" — таймер может предложить программе для Windows сохранять работу пользователя на диске всегда, когда истекает заданный интервал времени.
- Завершение демонстрационных версий программ — некоторые демонстрационные версии программ рассчитаны на свое завершение, скажем, через 30 минут после запуска. Таймер может сигнализировать таким приложениям, когда их время истекает.
- Задание темпа изменения — графические объекты в играх или окна с результатами в обучающих программах могут нуждаться в задании установленного темпа изменения. Использование таймера устраняет неритмичность, которая могла бы возникнуть из-за разницы в скоростях работы различных микропроцессоров.
- Мультимедиа — программы, которые проигрывают аудиодиски, звук или музыку, часто допускают воспроизведение звуковых данных в фоновом режиме. Программа может использовать таймер для периодической проверки объема воспроизведенной информации и координации его с информацией, выводимой на экран.

2.1 Основы использования таймера

Вы можете присоединить таймер к своей программе при помощи вызова функции *SetTimer*. Функция *SetTimer* содержит целый параметр, задающий интервал, который может находиться в пределах (теоретически) от 1 до 4 294 967 295 миллисекунд, что составляет около 50 дней. Это значение определяет темп, с которым Windows посылает вашей программе сообщения WM TIMER. Например, интервал в 1000 миллисекунд заставит Windows каждую секунду посылать вашей программе сообщение.

Если в вашей программе есть таймер, то она вызывает функцию *KillTimer* для остановки потока сообщений от таймера. Вы можете запрограммировать "однократный" таймер, вызывая функцию *KillTimer* при обработке сообщения WM_TIMER. Вызов функции *KillTimer* очищает очередь сообщений от всех необработанных сообщений WM_TIMER. После вызова функции *KillTimer* ваша программа никогда не получит случайного сообщения WM_TIMER.

2.1.1 Система и таймер

Таймер в Windows является относительно простым расширением таймерной логики, встроенной в аппаратуру PC и ROM BIOS. ROM BIOS компьютера инициализирует микросхему таймера так, чтобы она генерировала аппаратное прерывание. Это прерывание иногда называют "тиком таймера". Эти прерывания генерируются каждые 54.925 миллисекунды или примерно 18,2 раза в секунду. Некоторые программы, написанные для MS-

DOS, сами обрабатывают это аппаратное прерывание для реализации часов и таймеров.

В программах, сделанных для Windows, так не делается. Windows сама обрабатывает аппаратные прерывания и приложения их не получают. Для каждой программы, где в данный момент установлен таймер, Windows обрабатывает таймерное прерывание путем уменьшения на 1 значения счетчика, изначально переданного вызовом функции *SetTimer*. Когда это значение становится равным 0, Windows помещает сообщение WM_TIMER в очередь сообщений соответствующего приложения и восстанавливает начальное значение счетчика.

Поскольку приложения Windows получают сообщения WM_TIMER из обычной очереди сообщений, вам не нужно беспокоится о том, что ваша программа во время работы будет "прервана" внезапным сообщением WM_TIMER. В этом смысле таймер похож на клавиатуру и мышь: драйвер обрабатывает асинхронные аппаратные прерывания, а Windows преобразует эти прерывания в регулярные, структурированные, последовательные сообщения.

Таймер в Windows имеет ту же самую разрешающую способность 54.925 миллисекунды, что и встроенный таймер PC. Отсюда следуют два важных вывода:

- Приложение windows при использовании простого таймера не сможет ползать сообщения WM_TIMER в темпе, превышающем 18,2 раза в секунду.
- Временной интервал, который вы задаете при вызове функции *SetTimer,* всегда округляется вниз до целого числа кратного частоте срабатываний таймера. Например, интервал в 1000 миллисекунд, разделенный на 54.925 миллисекунды равен 18.207 срабатываниям таймера, которые округляются вниз до 18 срабатываний, что фактически составляет интервал в 989, а не 1000 миллисекунд. Для интервалов, меньших 55 миллисекунд, каждое срабатывание таймера генерирует одно сообщение WM_TIMER.

2.1.2 Таймерные сообщения не являются асинхронными

Как уже упоминалось, программы под DOS, написанные для IBM PC и совместимых компьютеров, могут использовать аппаратные срабатывания таймера, перехватывая аппаратное прерывание. Когда происходит аппаратное прерывание, выполнение текущей программы приостанавливается и управление передается обработчику прерываний. Когда прерывание обработано, управление возвращается прерванной программе.

Также как аппаратные прерывания клавиатуры и мыши, аппаратное прерывание таймера иногда называется асинхронным прерыванием, по-

скольку оно происходит случайно по отношению к прерываемой программе.

Хотя Windows тоже обрабатывает асинхронные таймерные прерывания, сообщения WM_TIMER, которые Windows посылает приложению, не являются асинхронными. Сообщения Windows ставятся в обычную очередь сообщений и обрабатываются как все остальные сообщения. Поэтому, если вы задаете функции *SetTimer* 1000 миллисекунд, то вашей программе не гарантируется получение сообщения WM_TIMER каждую секунду или даже каждые 989 миллисекунд. Если ваше приложение занято больше, чем секунду, то оно вообще не получит ни одного сообщения WM_TIMER в течение этого времени. Фактически, Windows обрабатывает сообщения WM_TIMER во многом также, как сообщения WM_PAINT. Оба эти сообщения имеют низкий приоритет, и программа получит только их, если в очереди нет других сообщений.

Сообщения WM_TIMER похожи на сообщения WM_PAINT и в другом смысле: Windows не хранит в очереди сообщений несколько сообщений WM_TIMER. Вместо этого Windows объединяет несколько сообщений WM_TIMER из очереди в одно сообщение. Поэтому, приложение не получает за раз группу сообщений WM_TIMER, хотя оно может получить два таких сообщения, быстро следующих одно за другим. В результате приложение не может определить число "потерянных" сообщений WM_TIMER.

2.2 Способы использования таймера

Если вам нужен таймер для измерения продолжительности работы вашей программы, вы, вероятно, вызовите *SetTimer* из функции *WinMain* или при обработке сообщения WM_CREATE, а *KillTimer* в ответ на сообщение WM_DESTROY. Установка таймера в функции *WinMain* обеспечивает простейшую обработку ошибки, если таймер недоступен. Вы можете использовать таймер одним из трех способов, в зависимости от параметров функции *SetTimer*.

Существует три способа использования таймера:

1 Получение сообщений WM_TIMER оконной процедурой

2 Вызов специальной процедуры обработки сообщений таймера

3 Способ для множественных вызовов

2.2.1 Получение сообщений WM_TIMER оконной процедурой

Этот простейший способ заставляет Windows посылать сообщения WM_TIMER обычной оконной процедуре приложения. Вызов функции *SetTimer* выглядит следующим образом:

```
SetTimer(hwnd, 1, iMsecInterval, NULL);
```

Первый параметр – это описатель того окна, чья оконная процедура будет получать сообщения WM_TIMER. Вторым параметром является идентификатор таймера, значение которого должно быть отличным от нуля. В этом примере он произвольно установлен в 1. Третий параметр — это 32-разрядное беззнаковое целое, которое задает интервал в миллисекундах. Значение 60000 задает генерацию сообщений WM_TIMER один раз в минуту.

Вы можете в любое время остановить поток сообщений WM_TIMER (даже во время обработки сообщения WM_TIMER), вызвав функцию:

```
KillTimer(hwnd, 1);
```

Вторым параметром здесь является тот же идентификатор таймера, который использовался при вызове функции *SetTimer*. Вы должны перед завершением вашей программы в ответ на сообщение WM_DESTROY уничтожить все активные таймеры.

Когда ваша оконная процедура получает сообщение WM_TIMER, значение *wParam* равно значению идентификатора таймера (который равен 1 в приведенном примере), а *lParam* равно 0. Если вам нужно более одного таймера, используйте для каждого таймера свой идентификатор. Значение параметра *wParam* позволит различать передаваемые в оконную процедуру сообщения WM_TIMER. Для того, чтобы вашу программу было легче читать, для разных идентификаторов таймера лучше использовать инструкции *#define:*

```
#define TIMER_SEC 1
#define TIMER_MIN 2
```

Два таймера можно задать, если дважды вызвать функцию *SetTimer*.

```
SetTimer (hwnd, TIMER_SEC, 1000, NULL);
SetTimer (hwnd, TIMER_MIN, 60000, NULL);
```

Логика обработки сообщения WM_TIMER может выглядеть примерно так:

```
case WM TIMER:
switch (wParam)
{
case TIMER SEC:
[обработка одного сообщения в секунду]
break;
case TIMER_MIN:
[обработка одного сообщения в минуту]
break;
}
return 0;
```

Если вы захотите установить новое время срабатывания для существующего таймера, уничтожьте таймер и снова вызовите функцию *SetTimer.* В этих инструкциях предполагается, что идентификатор таймера равен 1:

```
KiilTimer(hwnd, 1);
SetTimer (hwnd, I, iMsecInterval, NULL);
```

Параметр *iMsecInterval* задается равным новому времени срабатывания в миллисекундах. Вы можете использовать эту схему в программах часов, имеющих возможность выбора — показывать секунды или нет. Вам нужно просто изменить таймерный интервал с 1000 на 60 000 миллисекунд.

Если нет доступных таймеров, возвращаемым значением функции *SetTimer* является NULL. Ваша программа могла бы нормально работать и без таймера, но если таймер вам необходим, то у приложения не остается выбора, как только завершиться из-за невозможности работать. Если вы вызываете в *WinMain* функцию *SetTimer,* то вы можете завершить свою программу, просто возвращая FALSE из *WinMain.*

Предположим, что вам нужен 1000-миллисекундный таймер. Сразу за вызовом *CreateWindow,* но перед циклом обработки сообщений, вы могли бы вставить следующую инструкцию:

```
if(!SetTimer(hwnd, 1, 1000, NULL))
return FALSE;
```

Но это некрасивый способ завершения программы. Пользователь останется в неведении, почему не загружается его приложение. Гораздо удобнее — и намного проще — для вывода сообщения на экран использовать окно сообщений Windows

Окно сообщений — это всплывающее окно, которое появляется всегда в центре экрана. В окнах сообщений есть строка заголовка, но нет рамки, позволяющей изменять размеры окна. Строка заголовка обычно содержит имя приложения. Окно сообщений включает в себя само сообщение и одну, две или три кнопки (какие-либо сочетания кнопок OK, Retry, Cancel, Yes, No и других). В окне сообщений может также находиться ранее определенный значок: строчное "i" (что означает "information" - информация), восклицательный, вопросительный или запрещающий знаки. Последний представляет собой белый символ X на красном фоне (как настоящий знак "стоп"). Вы, вероятно, уже видели множество окон сообщений при работе с Windows.

Этот код создает окно сообщений, которое вы можете использовать, когда функция *SetTimer* терпит неудачу при установке таймера:

```
if(!SetTimer(hwnd, 1, 1000, NULL))
```

```
{
   MessageBox (hwnd,"Too many clocks or timers!", "Program Name",
MB_ICONEXCLAMATION | MB_OK);
   return FALSE;
}
```

Когда пользователь нажимает клавишу <Enter> или щелкает на кнопке OK, WinMain завершается и возвращает значение FALSE. По умолчанию окна сообщений являются "модальными окнами приложения" (application modal). Это означает, что пользователь должен как-то отреагировать на окно сообщений перед тем, как приложение продолжит работу. Однако, пользователь может переключиться на другие приложения. Следовательно, почему бы не дать пользователю возможность закрыть одно из использующих таймер приложений и тогда уже успешно загрузить свое приложение с помощью следующего кода:

```
while(!SetTimer(hwnd„ 1, 1000, NULL))
   if(IDCANCEL == MessageBox(hwnd, "Too many clocks or timers!",
     "Program Name", MB_ICONEXCLAMATION | MB_RENTRYCANCEL) )
      return FALSE;
```

Если пользователь щелкает на кнопке Cancel, функция *MessageBox* возвращает значение равное IDCANCEL, и программа завершается. Если пользователь щелкает на кнопке Retry, функция *SetTimer* вызывается снова.

Рассмотрим пример программы устанавливающей таймер на временной интервал, равный 1 секунде. При получении сообщения WM_TIMER, программа изменяет цвет рабочей области с голубого на красный или с красного на голубой и, вызывая функцию *MessageBeep,* издает гудок. Хотя в документации функция *MessageBeep* описывается в связи с функцией *MessageBox,* реально она всегда может использоваться для звукового сигнала. В компьютерах, оборудованных звуковой платой, можно использовать различные параметры MB_ICON в функции *MessageBeep* для воспроизведения разнообразных звуков.

В программе таймер устанавливается в функции *WinMain,* а сообщения WM_TIMER обрабатываются в оконной процедуре *WndProc*. При обработке сообщения WM_TIMER программа вызывает функцию *MessageBeep,* инвертирует значение *bFlipFlop* и, для генерации сообщения WM_PAINT, делает окно недействительным. При обработке сообщения WM_PAINT программа с помощью вызова функции *GetClientRect* получает структуру RECT, соответствующую размерам окна целиком, и с помощью вызова функции *FillRect,* закрашивает окно.

```
BEEPER1.C
#include <windows.h>
```

```
#define ID_TIMER    1
#pragma argsused
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static char szAppName[] = "Beeper1" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASS  wndclass ;
/*     wndclass.cbSize        = sizeof (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 ;
/*     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ; */
     RegisterClass (&wndclass) ;
     hwnd = CreateWindow (szAppName, "Beeper1 Timer Demo",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          CW_USEDEFAULT, CW_USEDEFAULT,
                          NULL, NULL, hInstance, NULL) ;
  while (!SetTimer (hwnd, ID_TIMER, 1000, NULL))
    if (IDCANCEL == MessageBox (hwnd,
                          "Too many clocks or timers!", szAppName,
                            MB_ICONEXCLAMATION | MB_RETRYCANCEL))
        return FALSE ;
  ShowWindow (hwnd, iCmdShow) ;
  UpdateWindow (hwnd) ;
  while (GetMessage (&msg, NULL, 0, 0))
  {
    TranslateMessage (&msg) ;
    DispatchMessage (&msg) ;
  }
  return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM
lParam)
{
  static BOOL fFlipFlop = FALSE ;
  HBRUSH      hBrush ;
  HDC         hdc ;
  PAINTSTRUCT ps ;
  RECT        rc ;

  switch (iMsg)
  {
```

```
    case WM_TIMER :
      MessageBeep (0) ;
      fFlipFlop = !fFlipFlop ;
      InvalidateRect (hwnd, NULL, FALSE) ;
      return 0 ;

    case WM_PAINT :
      hdc = BeginPaint (hwnd, &ps) ;
      GetClientRect (hwnd, &rc) ;
      hBrush = CreateSolidBrush (fFlipFlop ? RGB(255,0,0) :
RGB(0,0,255)) ;
      FillRect (hdc, &rc, hBrush) ;
      EndPaint (hwnd, &ps) ;
      DeleteObject (hBrush) ;
      return 0 ;

    case WM_DESTROY :
      KillTimer (hwnd, ID_TIMER) ;
      PostQuitMessage (0) ;
      return 0 ;
  }
  return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}
```

2.2.2 Вызов процедуры обработки сообщений таймера

При первом способе установки таймера сообщения WM_TIMER посылаются в обычную оконную процедуру. С помощью второго способа вы можете заставить Windows пересылать сообщение таймера другой функции из вашей программы.

Функция, которая будет получать эти таймерные сообщения, называется функцией "обратного вызова". Это функция вашей программы, которую вызывает Windows. Вы сообщаете Windows адрес этой функции, а позже Windows вызывает ее. Это должно быть вам знакомым, поскольку оконная процедура программы фактически является такой функцией обратного вызова. При регистрации класса окна вы сообщаете Windows адрес оконной процедуры, и затем Windows, посылая сообщения программе, вызывает эту функцию.

SetTimer — это не единственная функция Windows, использующая функцию обратного вызова.

Также как и оконная процедура, функция обратного вызова должна определяться как CALLBACK, поскольку Windows вызывает ее вне кодового пространства программы. Параметры функции обратного вызова и ее возвращаемое значение зависят от назначения функции обратного вызова. В случае функции обратного вызова для таймера, входными параметрам1 являются те же параметры, что и параметры оконной процедуры. Таймерная функция обратного вызова не имеет возвращаемого в Windows значения.

Давайте назовем функцию обратного вызова *TimerProc*. (Вы можете дать ей любое имя). Она будет обрабатывать только сообщения WM_TIMER.

```
VOID CALLBACK TimerProc (HWND hwnd, _UINT iMsg, _UINT iTimerlD, DWORD
dwTime)
{
[обработка сообщений WM_TIMER]
}
```

Входной параметр *hwnd* — это описатель окна, задаваемый при вызов функции *SetTimer*. Windows будет посылать функции *TimerProc* только сообщения WM_TIMER, следовательно параметр *iMsg* всегда будет равен WM_TIMER. Значение *iTimerID* — это идентификатор таймера, а значение *dwTime* — системное время.

Как уже говорилось выше, для первого способа установки таймера требуется следующий вызов функции *SetTimer*.

```
SetTimer(hwnd, iTimerlD, iMsecInterval, NULL);
```

При использовании функции обратного вызова для обработки сообщений WM_TIMER, четвертый параметр функции *SetTimer* заменяется адресом функции обратного вызова:

```
SetTimer (hwnd, iTimerlD, iMsecInterval, (TIMERPROC) TimerProc);
```

Пример программы

Давайте рассмотрим пример программы, чтобы вы могли увидеть, как это все работает

```
BEEPER2.C
#include <windows.h>
#define ID_TIMER    1
LRESULT CALLBACK WndProc   (HWND, UINT, WPARAM, LPARAM) ;
VOID    CALLBACK TimerProc (HWND, UINT, UINT,   DWORD ) ;

#pragma argsused
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static char szAppName[] = "Beeper2" ;
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASS  wndclass ;
/*     wndclass.cbSize        = sizeof (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 ;
/* wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ; */
  RegisterClass (&wndclass) ;
  hwnd = CreateWindow (szAppName, "Beeper2 Timer Demo",
                       WS_OVERLAPPEDWINDOW,
                       CW_USEDEFAULT, CW_USEDEFAULT,
                       CW_USEDEFAULT, CW_USEDEFAULT,
                       NULL, NULL, hInstance, NULL) ;
  while (!SetTimer (hwnd, ID_TIMER, 1000, (TIMERPROC) TimerProc))
    if (IDCANCEL == MessageBox (hwnd,
                          "Too many clocks or timers!", szAppName,
                          MB_ICONEXCLAMATION | MB_RETRYCANCEL))
              return FALSE ;
  ShowWindow (hwnd, iCmdShow) ;
  UpdateWindow (hwnd) ;
  while (GetMessage (&msg, NULL, 0, 0))
  {
    TranslateMessage (&msg) ;
    DispatchMessage (&msg) ;
  }
  return msg.wParam ;
}
LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM
lParam)
{
  switch (iMsg)
  {
    case WM_DESTROY :
      KillTimer (hwnd, ID_TIMER) ;
      PostQuitMessage (0) ;
      return 0 ;
  }
  return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}

#pragma arsused
VOID CALLBACK TimerProc (HWND hwnd, UINT iMsg, UINT iTimerID, DWORD
dwTime)
{
  static BOOL fFlipFlop = FALSE ;
  HBRUSH      hBrush ;
  HDC         hdc ;
  RECT        rc ;

  MessageBeep (0) ;
  fFlipFlop = !fFlipFlop ;
  GetClientRect (hwnd, &rc) ;
  hdc = GetDC (hwnd) ;
```

```
    hBrush = CreateSolidBrush (fFlipFlop ? RGB(255,0,0) : RGB(0,0,255))
;
    FillRect (hdc, &rc, hBrush) ;
    ReleaseDC (hwnd, hdc) ;
    DeleteObject (hBrush) ;
}
```

2.2.3 Способ для множественных вызовов

Третий способ установки таймера напоминает второй, за исключением того, что параметр *hwnd* функции *SetTimer* устанавливается в NULL, а второй параметр (обычно идентификатор таймера) игнорируется. Функция возвращает ID таймера:

```
iTimerlD=SetTimer (NULL, 0, wMsecInterval, (TIMERPROC) TimerProc);
```

Возвращаемое функцией *SetTimer* значение *iTimerlD* будет равно NULL, если таймер недоступен.

Первый параметр функции *KillTimer* (обычно описатель окна) также должен быть равен NULL. Идентификатор таймера должен быть равен значению возвращаемому функцией *SetTimer.*

```
KillTimer (NULL, iTimerlD);
```

Параметр *hwnd,* передаваемый в *TimerProc,* также должен быть равен NULL.

Такой метод установки таймера используется редко. Он удобен, если в программе в разное время делается много вызовов функции *SetTimer,* и при этом не запоминаются те таймерные идентификаторы, которые уже использовались.

3 Контрольные вопросы

1) Какое сообщение посылает системный таймер?
2) Какая функция предназначена для установки таймера?
3) Какая функция предназначена для удаления таймера?
4) Как работает процедура обратного вызова?
5) Какая частота у системного таймера?

4 Задание

1) Изучить описание лабораторной работы.
2) Составить приложение Windows, выполняющее некоторые регулярные действия, через произвольный промежуто времени максимально используя при этом системный таймер.
3) Создать программу реализующую часы.
4) Отладить и протестировать полученную программу.
5) Оформить отчёт.

Список литературы

1. Гордеев А. В., Молчанов А. Ю. Системное программное обеспечение. – СПб.: Питер, 2001. – 736 с.: ил.
2. Харт, Д. М. Системное программирование в среде Win32, 2-е изд.: Пер. с англ.: – М.: Издательский дом «Вильямс», 2001.– 464 с.: ил. – Парал. тит. англ.
3. Петзольд Ч. Программирование для Windows 95; в двух томах. Том 1/ Пер. С англ. — СПб.: BHV— Санкт-Петербург, 1997.— 752 с.: ил.
4. Петзольд Ч. Программирование для Windows 95; в двух томах. Том 2/ Пер. С англ. — СПб.: BHV— Санкт-Петербург, 1997.— 368 с.: ил.
5. Румянцев П.В. Азбука программирования в Win 32 API. – М.: Горячая Линия - Телеком, 2004. – 312 с.
6. Ганеев Р. М. Проектирование интерфейса пользователя средствами Win32 API. – М: Горячая Линия – Телеком, 2001. – 336 с.
7. Харт, Д. М. Системное программирование в среде Windows Windows System Programming. – М.: Вильямс, 2005.– 336 с.: ил.
8. Неббет, Гэри. Справочник по базовым функциям API Windows NT/2000 Windows NT/2000 – М: Вильямс, 2002. – 528 с.

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ
Часть 1. Основы работы с WinAPI

Методические указания

Составители: Мурлин Алексей Георгиевич;
Волик Александр Георгиевич;

Авторская правка

Компьютерная верстка А. Г. Волик

Подписано в печать Изд. № 000
Электронное издание Изд. каф.
0,9 Мб

Кубанский государственный технологический университет
350072, г. Краснодар, ул. Московская, 2, кор. А
Типография КубГТУ: 350058, г. Краснодар, ул. Старокубанская, 88/4