Министерство образования и науки
Российской Федерации

Государственное образовательное учреждение
высшего профессионального образования
«Алтайский государственный технический
университет им. И.И.Ползунова»

Л.И.Сучкова

Win32 API:
основы программирования

Барнаул – 2010

УДК 004.42

Сучкова, Л.И. Win32 API: основы программирования: учебное пособие/ Л.И. Сучкова; АлтГТУ им. И.И. Ползунова. – Барнаул, АлтГТУ, 2010. – 138 с., ил.

В учебном пособии изложены основы логики обработки сообщений в Windows, рассмотрена структура базового приложения, а также API-функции, обеспечивающие графический интерфейс, работу с элементами управления форм, работу с динамическими библиотеками. Для каждой темы рассмотрены программы - примеры выполнения заданий.

Учебное пособие предназначено для студентов, обучающихся по направлениям 230100 «Информатика и вычислительная техника», 230110 «Программная инженерия».

Учебное пособие рассмотрено на совместном заседании кафедр «Прикладная математика» и «Вычислительные системы и информационная безопасность», протокол № 10 от 26.06.2010 г.

Рецензент: А.С. Шатохин, к.т.н., профессор, проректор по информатизации АлтГУ

Содержание

1 МИНИМАЛЬНОЕ ОКОННОЕ ПРИЛОЖЕНИЕ

1.1 Сообщения и их структура

Программирование в Windows основывается на использовании интерфейса прикладного программирования **API** (**A**pplication **P**rogram **I**nterface). API предоставляют программисту набор готовых классов, функций, структур и констант. Их количество составляет около двух тысяч. API-функции обеспечивают взаимодействие приложения с внешними устройствами и ресурсами операционной системы.

Главным элементом оконного приложения в среде Windows является **окно**, которое может содержать элементы управления: кнопки, списки, окна редактирования, полосы прокрутки и т.п. Эти элементы также являются окнами, обладающими особыми свойствами. События, происходящие с элементами управления и с самим окном, приводят к формированию **сообщений** - специально оформленных групп данных.

Приложение не знает порядка появления сообщений, поэтому оно должно быть построено таким образом, чтобы обеспечить корректную и предсказуемую работу при поступлении сообщений любого типа – системных или пользовательских.

Отличительным признаком сообщения является его код, который для системных сообщений лежит в диапазоне от 1 до 0x3FF. Так как с кодами работать в программе неудобно, то каждому коду сопоставляется своя символическая константа, по имени которой можно определить источник сообщения. Например, при перемещении мыши возникает сообщение WM_MOUSEMOVE (код 0x200), при нажатии на левую кнопку мыши - сообщение WM_LBUTTONDOWN (код 0x201). При перерисовке окно получает сообщение WM_PAINT. Эти события относятся к классу аппаратных, поскольку в их обработке участвуют драйверы внешних устройств. Например, при нажатии клавиши драйвер клавиатуры формирует пакет данных и пересылает его в форме сообщения в системную очередь сообщений Windows. Рассмотрим дальнейшую судьбу сообщения.

Для 32-разрядных приложений время для выполнения распределяется между потоками приложения, как минимум, приложение создает один поток. Для каждого потока создается своя очередь сообщений, которая не имеет фиксированного размера. Сообщения из системной очереди распределяются между очередями сообщений потоков, откуда затем извлекаются приложением с помощью функции **GetMessage()**.

Возникает вопрос, в какую очередь потока направляется сообщение из системной очереди? Предположим, что на экране много окон при-

ложений и возникло событие, инициированное мышью. В этом случае сообщение от мыши будет адресованы потоку-владельцу окна, над которым находится курсор, т.к. при помощи мыши логично активизировать именно действия, связанные с текущим окном. А значит, сообщения от мыши ставятся в очередь потока, активизировавшего текущее окно.

Источниками сообщений, помимо устройств, могут быть прикладные программы или ОС. Если сообщение создается в прикладной программе и посылается в Windows, чтобы ОС выполнила требуемые действия, то такое сообщение имеет код, превышающий 0x3FF. Программист может также предусмотреть собственные сообщения и направлять их в различные окна приложения для оповещения о различных ситуациях в вычислительной системе.

Рассмотрим структуру одного сообщения. Сообщения передаются в приложение с помощью специальной структуры **MSG**, включающей 6 полей, и описанной в файле *winuser.h*. Так как язык программирования практически не влияет на общую структуру Windows-приложения, рассмотрим реализацию на С. Поля структуры **MSG**:

```
{
HWND hwnd;  // Дескриптор окна, которому адресовано
            // сообщение
UINT message;  // Код данного сообщения
WPARAM wParam; // Доп. информация
LPARAM lParam; // Доп. информация
DWORD time;    // Время отправления сообщения
POINT pt; // Позиция курсора мыши на момент
          // отправления сообщения
} MSG;
```

Все объекты (окна, файлы, процессы, потоки, события и т.д.) и системные ресурсы в Windows описываются с помощью дескрипторов. Различают, например, дескрипторы экземпляров приложений (Handle of Instance, HInstance), окон (HWND), пиктограмм (HIcon), шрифтов(HFont), перьев (HPen) и т.д. Определения этих дескрипторов доступны при подключении W*indows.h*, который ссылается на *Windef.h*. Так как сообщение посылается определенному окну, то его дескриптор указывается в структуре сообщения. Поля **wParam** и **lParam** содержат дополнительные данные, необходимые для обработки сообщения, их содержимое различается для сообщений каждого типа. Например, для сообщения WM_MOUSEMOVE поле **wParam** содержит информацию о состоянии клавиш мыши (нажаты или отпущены), а также о состоянии клавиш Ctrl и Shift, а поле **lParam** содержит позицию курсора относительно начала клиентской области окна в отличие от поля **pt**, со-

держащего позицию курсора мыши относительно границ экрана (рисунок 1.1).

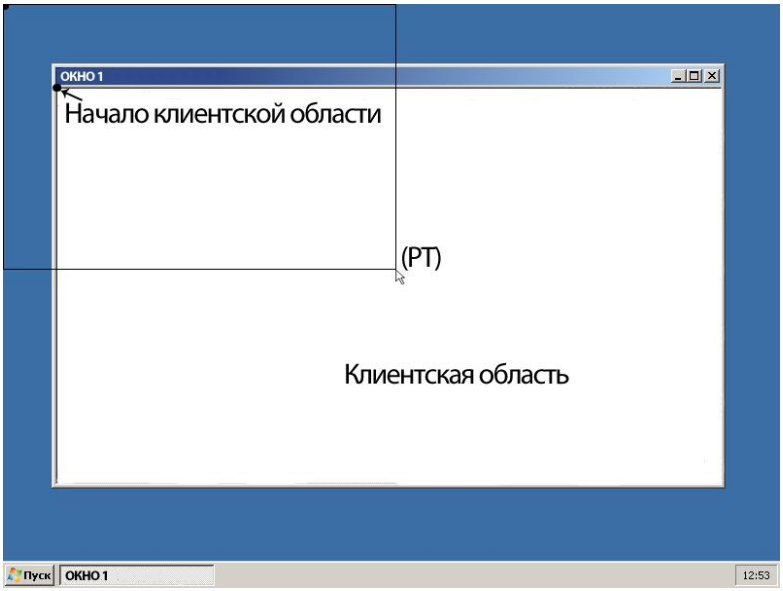


Рисунок 1.1 – Окно с минимальным набором элементов управления

Отметим, что приложение может обрабатывать не все сообщения, часть из них обрабатываются самой ОС либо до попадания сообщения в очередь потока, либо после извлечения сообщения из очереди и «осознания», что приложение сообщение данного типа не интересует. Рассмотрим первый из этих случаев. Например, если щелкнуть левой кнопкой мыши на пункте меню, то вместо сообщения WM_LBUTTONDOWN формируется сообщение WM_COMMAND, параметры которого содержат идентификатор пункта меню, над которым был курсор мыши в момент щелчка. Это избавляет приложение от необходимости анализа положения курсора мыши при оценке, попадает ли курсов в прямоугольную область, соответствующую пункту меню.

При программировании под Windows необходимо знать правила наименования различных объектов. В настоящее время для конструирования имен объектов используется система венгерской записи, согласно которой перед именем объекта ставятся символы, по которым

можно определить тип переменной. Эти символы называются префиксом. Так, префикс **sz** (**S**tring **Z**ero) означает символьную строку, заканчивающуюся двоичным нулем, **c** (**C**har) – символ, **dw** (**D**ouble **W**ord) - 32 битная переменная, **lpsz** (**L**ong **P**ointer of **S**tring **Z**ero)– дальний (на самом деле ближний!) указатель на символьную строку, заканчивающуюся двоичным нулем, **h** (**H**andle) – дескриптор (описатель) объекта, содержащий информацию об объекте.

Сообщения от Windows имеют префикс WM_, префикс BM_ соответствует сообщениям от кнопок, префикс EM_ - сообщениям от текстовых полей редактирования (EditBox), LB_ - сообщениям от списков.

1.2. Работа с окном и организация цикла обработки сообщений в главной функции WinMain

1.2.1. Оконные сообщения и функции работы с окнами

Окно – это не только область на экране, посредством которой приложение может представить свой вывод, это еще и адресат событий и сообщений в среде Windows.

Окно идентифицируется по ***дескриптору окна***. Этот дескриптор (переменная типа HWND) однозначно определяет каждое окно в системе. Windows организует свои окна в иерархическую структуру:

- каждое окно имеет родителя, корнем дерева всех окон является окно рабочего стола, создаваемого Windows при загрузке;

- для всех окон верхнего уровня (для главных окон приложений и других перекрывающихся и всплывающих окон приложений) родительским окном является рабочий стол.

Родитель дочернего окна – окно верхнего уровня или другое дочернее окно, более высокого уровня по иерархии.

Между окнами верхнего уровня (перекрывающиеся и всплывающие окна) существует еще одна иерархическая связь. Владельцем окна верхнего уровня может быть другое окно того же уровня. Окно, имеющее владельца, всегда отображается поверх своего владельца и исчезает при минимизации окна-владельца. Типичным случаем владения одного окна верхнего уровня другим является приложение, отображающее диалоговое окно. Диалоговое окно не является дочерним окном (оно является всплывающим окном), но его владельцем остается окно приложения.

Окно, как правило, реагирует на множество сообщений. Рассмотрим *наиболее часто обрабатываемые сообщения*:

- WM_CREATE - посылается окну перед тем, как окно станет видимым, при получении сообщения приложение может инициализировать нужные данные;

- WM_DESTROY - посылается окну, которое уже удалено с экрана и должно быть разрушено;

- WM_CLOSE - указывает, что окно должно быть закрыто. Приложение может при его обработке, например, вывести диалоговое окно подтверждения закрытия окна;

- WM_QUIT – сообщение, требующее завершить приложение;

- WM_QUERYENDSESSION - уведомляет приложение о намерении Windows закончить сеанс. Приложение может возвратить значение FALSE в ответе на это сообщение, предотвратив этим выключение Windows. После обработки сообщения WM_QUERYENDSESSION Windows посылает всем приложениям сообщение WM_ENDSESSION с результатами этой обработки;

- WM_ENDSESSION - посылается приложениям после обработки сообщения WM_QUERYENDSESSION. Оно указывает, должна ли Windows выключиться, или выключение отложено. При указании выключения сеанс Windows может закончиться в любое время после обработки сообщения WM_ENDSESSION всеми приложениями. Поэтому важно, чтобы приложения выполнили все задачи по обеспечению безопасного завершения работы;

- WM_ACTIVATE - указывает, что окно верхнего уровня будет активизировано или деактивизировано. Сообщение сначала посылается окну, которое должно быть деактивизировано, а потом окну, которое должно быть активизировано;

- WM_SHOWWINDOW - указывает, что окно должно быть скрыто или отображено;

- WM_ENABLE – посылается окну, когда оно становится доступным или недоступным. Недоступное окно не может принимать вводимые данные от мыши или клавиатуры;

- WM_MOVE – указывает, что расположение окна изменилось;

- WM_SIZE – указывает, что размер окна был изменен;

- WM_SETFOCUS – указывает получение окном фокуса клавиатуры;

- WM_KILLFOCUS – указывает, что окно должно потерять фокус клавиатуры;

Рассмотрим функции, позволяющие приложению исследовать иерархию окон, находить, перемещать, изменять режим отображения, изменять вид окна:

AnimateWindow - дает возможность производить специальные эф-

факты при показе или сокрытии окон. Имеются четыре типа мультипликации: ролик, слайд, свертывание или развертывание и плавное перетекание;

CloseWindow - свертывает (но не разрушает) определенное окно;

FindWindow – используется для поиска окна верхнего уровня по имени его класса окна или по заголовку окна;

FlashWindow - предназначена для создания окна с мигающим заголовком, используется для привлечения внимания к окну;

FlashWindowEx – усовершенствованный вариант FlashWindow;

GetClientRect - возвращает координаты клиентской области окна;

GetParent – возвращает дескриптор родительского окна для указанного;

GetDesktopWindow - возвращает дескриптор окна рабочего стола Windows;

GetTitleBarInfo - возвращает информацию о строке заголовка;

GetWindow – предоставляет наиболее гибкий способ работы с иерархией окон. В зависимости от значения второго параметра эту функцию можно использовать для получения идентификатора родительского окна, владельца, окон одного уровня или дочерних окон.

GetWindowPlacement - возвращает данные о расположении окна;

GetWindowTextLength - возвращает длину (количество символов) текста строки заголовка для окна, если окно имеет область заголовка. Если окно - элемент управления, функция возвращает длину текста внутри элемента управления.

IsChild - проверяет, является ли окно дочерним окном или порожденным окном для указанного родительского окна;

IsWindow - определяет, соответствует ли заданный дескриптор существующему окну;

IsWindowVisible – возвращает информацию о состоянии заданного окна;

MoveWindow - изменяет расположение и размеры окна. Для окна верхнего уровня расположение вычисляется относительно левого верхнего угла экрана. Для дочернего окна расположение вычисляется относительно левого верхнего угла клиентской области родительского окна;

OpenIcon - восстанавливает свернутое окно;

SetWindowPlacement - устанавливает в состояние показа и восстанавливает, свертывает и развертывает окно;

SetWindowText - копирует текст строки заголовка окна (если оно имеет ее) в буфер. Если окно - элемент управления, текст элемента управления копируется;

ShowWindow - устанавливает состояние показа окна;

WindowFromPoint - отыскивает дескриптор окна, которое содержит заданную точку.

1.2.2. Структура главной функции в минимальном приложении

Минимальное приложение Windows состоит из 2 частей:

1. главной функции с именем **WinMain**, включающей цикл обработки сообщений;
2. оконной функции **WndProc**.

Выполнение любого оконного приложения начинается с главной функции. Она содержит код, осуществляющий инициализацию приложения в операционной системе, с помощью которого система узнает о новом приложении и его свойствах. Для этого в **WinMain** описывается и регистрируется класс окна приложения, а затем создается и отображается на экране окно приложения зарегистрированного класса. Видимым для пользователя результатом работы главной функции является появление на экране нового графического объекта - окна. Последним действием кода главной функции является создание цикла обработки сообщений. После его создания приложение начинает взаимодействовать с вычислительной системой через сообщения.

Обработка же поступающих приложению сообщений осуществляется с помощью специальной функции, называемой оконной. Оконная функция уникальна тем, что может быть вызвана только из операционной системы, а не из приложения, которое ее содержит.

Рассмотрим структуру главной функции более подробно. На С минимальная главная функция с одним окном имеет вид:

```
int WINAPI WinMain(HINSTANCE hInst, HINSTANCE, LPSTR,
int)
   { char szClassName[]="Class32";
   // Произвольное имя класса главного окна
     char szTitleName[]="Пример приложения";
   // Произвольный  заголовок окна
    MSG mess; // Структура для получения сообщений
    WNDCLASS wc; //Структура для задания характеристик
                 //окна
   // Регистрация класса главного окна
    memset(&wc,0,sizeof(wc));
   // Обнуление и формирование необходимых элементов
   // структуры wc
    wc.lpfnWndProc=WndProc; // указатель на оконную
```

```
                                //функцию
    wc.hInstance=hInst; // Дескриптор приложения
    wc.hIcon=LoadIcon(NULL,IDI_APPLICATION);
   //вызов функции загрузки стандартной пиктограммы
    wc.hCursor=LoadCursor(NULL,IDC_ARROW);
   // вызов функции загрузки курсора мыши –
   // однонаправленной стрелки
    wc.hbrBackground=GetStockBrush(LTGRAY_BRUSH);
   // запрос для закрашивания фона окна
   // светло-серой кистью из ресурсов Windows
    wc.lpszClassName=szClassName; // имя класса окна
    RegisterClass(&wc); // Вызов функции регистрации
                        //класса окна
   // Создадим главное окно
    HWND hwnd=CreateWindow(szClassName,szTitleName,
   // Класс и заголовок окна
    WS_OVERLAPPEDWINDOW,20,20,300,200,
   // Стиль, x и y координаты левого верхнего угла,
   // ширина и высота окна
   HWND_DESKTOP,NULL,hInst,NULL);
   // Дескриптор родительского окна, ссылка на меню,
   //  Дескриптор экземпляра приложения, адрес дополни-
тельных данных
   // Покажем окно на экране
    ShowWindow(hwnd,SW_SHOWNORMAL);
   // Организуем цикл обработки сообщений
    while(GetMessage(&mess,NULL,0,0))
   // получить сообщение
       DispatchMessage(&mess);
   // и передать оконной функции для обработки
    return 0;
   // после выхода из цикла вернуться в Windows
   }
```

При запуске приложения управление передается программам ОС, которые загружают код приложения в память и вызывают главную функцию, которая должна иметь имя **WinMain** и дескриптор WINAPI. При вызове WinMain ОС передает ей 4 параметра. Первый параметр типа HINSTANCE должен быть дескриптором данного экземпляра приложения, необходимым для его идентификации. Этот дескриптор используется различными API-функциями, поэтому он должен храниться в глобальной переменной. В минимальном приложении таких функций нет, поэтому **hInst** не сохраняется нигде. Второй параметр для 16-разрядных приложений представляет собой дескриптор предыдущего экземпляра приложения, а для 32-разрядных приложений не

имеет смысла и принимает нулевое значение. Третий параметр является указателем на строку, содержащую параметры командной строки при запуске приложения. Четвертый параметр **WinMain** служит для определения режима запуска приложения. Так, приложение можно сразу после запуска свертывать в пиктограмму, что осуществляется при равенстве 4 параметра 7. Если 4 параметр равен 1, то окно приложения принимает размеры, указанные при создании дескриптора окна. Этот 4 параметр обычно передается вторым параметром функции показа окна **ShowWindow**. В нашем примере мы сразу передали функции **ShowWindow** нужную константу SW_SHOWNORMAL, равную 1.

В приложении главная функция должна сначала зарегистрировать в ОС класс главного окна. Если кроме одного главного окна планируется вывод на экран дочерних окон, то их классы также необходимо зарегистрировать. **ОС выводит на экран и обслуживает только окна зарегистрированных классов!** Класс окна задает характеристики, общие для всех подобных окон. Действия программиста по регистрации класса заключаются в заполнении структуры типа WNDCLASS, в которой хранятся характеристики класса окна. Данные из этой структуры необходимы функции **RegisterClass**, которая и выполняет собственно регистрацию. Рассмотрим структуру WNDCLASS.

```
{
UINT style; // стиль класса окна
WNDPROC lpfnWndProc;
// указатель на имя оконной функции
int cbClsExtra;
// количество байтов дополнительной информации
// о классе
int cbWndExtra;
// количество байтов дополнительной информации
// об окне
HINSTANCE hInstance; // дескриптор приложения
HICON hIcon; // дескриптор пиктограммы приложения
HCURSOR hCursor; // дескриптор курсора приложения
HBRUSH hbrBackground; // дескриптор кисти для фона ок-
на
LPCSTR lpszMenuName;
// указатель на строку с именем меню окна
LPCSTR lpszClassName;
// указатель на строку с именем класса окна
} WNDCLASS;
```

Не все поля структуры необходимо заполнять, поэтому структура сначала обнуляется, а затем заполняются только нужные поля. Нуле-

вое значение элемента структуры, описывающей объект Windows (например, структуры типа WNDCLASS), означает, что реальное значение элемента будет установлено по умолчанию. Через структуру WNDCLASS ОС определяет адрес оконной функции, которая вызывается для обработки сообщений.

Стиль класса окна представляет собой целое 32-битовое число, каждый бит которого разрешает или запрещает возможность определенных действий с окном. Так, 0 и 1 биты разрешают/запрещают перерисовку окна при изменении его размеров, бит 3 отвечает за реакцию приложения на двойные щелчки мышью в области окна, бит 9 запрещает закрытие окна пользователем. За каждый бит или их комбинацию отвечает предопределенная константа, имя которой может быть использовано для формирования стиля класса окна с операцией побитового ИЛИ. Примерами констант, определяющих стиль класса окна, являются CS_HREDRAW (перерисовка окна при изменении его ширины), CS_VREDRAW (перерисовка окна при изменении высоты), CS_NOCLOSE (запрещение команды закрытия окна в системном меню). В нашем минимальном приложении создание своего стиля окна не требуется.

Курсор и пиктограмма являются ресурсами Windows. Ресурсы обычно загружаются из файла ресурсов с помощью специальных функций загрузки ресурсов, например, для загрузки курсора используется **LoadCursor**. В качестве первого аргумента функции загрузки ресурсов указывается дескриптор приложения, в котором хранится требуемый ресурс, а в качестве второго – имя ресурса. При этом необязательно пользоваться ресурсами приложений, можно использовать ресурсы Windows. Для этого первым параметром функции **LoadCursor** указывается NULL, что означает саму Windows, а второй параметр выбирается из списка встроенных ресурсов. Так, для курсора-однонаправленной стрелки имя ресурса – IDC_ARROW, для курсора вертикальной двунаправленной стрелки – IDC_SIZENC. С помощью функции **LoadIcon** загружается пиктограмма приложения, также выбираемая из стандартных ресурсов Windows. Для формирования кисти, определяющей закраску фона окна, используется одна из предопределенных кистей Windows. Можно установить свою кисть, использовав вызов вида:

```
wc.hbrBackground=CreateSolidBrush(RGB(100,70,150));
```

где аргументом **CreateSolidBrush** является цвет кисти, упаковываемый в 4 байта с помощью макроса **RGB**, аргументами которой служат интенсивности трех составляющих цвета.

После регистрации класса окна необходимо создать главное окно и показать его на экране. Порожденные окна тоже необходимо перед показом создать, причем необязательно в **WinMain**. Создание окна осуществляется с помощью функции **CreateWindow**, имеющей следующие параметры:

- адрес строки с именем регистрируемого класса. Отметим, что имя класса может быть предопределенным для окон, реализующих элементы управления. Класс BUTTON служит для формирования кнопок, групп, флажков и переключателей, класс LISTBOX служит формирования списка строк, класс EDIT – для формирования поля редактирования при вводе текста, класс STATIC – для размещения в окне текста, класс SCROLLBAR – для формирования полосы прокрутки;
- адрес строки с заголовком окна;
- стиль окна, определяющий вид окна при показе, например, вид рамки, наличие заголовка, кнопок, рамки и т.п. Стиль – это 32-битное число, биты которого несут смысловую нагрузку по определению конкретных свойств окна при отображении. Для того чтобы указать для окна набор свойств, используют побитовое ИЛИ для констант, определяющих то или иное свойство. Обычно главное окно описывается с помощью константы WS_OVERLAPPEDWINDOW (0x00CF0000h), которая одновременно указывает на возможность перекрытия окна, наличие строки заголовка, системного меню, рамки, кнопки свертывания и кнопки развертывания окна, что обеспечивается операцией побитового ИЛИ между константами WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX, WS_MAXIMIZEBOX;
- **x** и **y** координаты левого верхнего угла окна относительно начала экрана;
- размеры окна в пикселах по горизонтали и вертикали;
- дескриптор родительского окна. Для главного окна будем считать, что его родителем является рабочий стол, имеющий дескриптор HWND_DESKTOP;
- дескриптор меню окна. Если его нет или используется меню класса, указывается NULL;
- Дескриптор приложения, полученный через первый аргумент функции **WinMain**;
- адрес дополнительных данных, необходимых для создания окна.

Если создание окна прошло успешно, то функция **CreateWindow** возвращает дескриптор созданного окна, который передается в функцию **ShowWindow**, выводящую окна на экран.

В конце функции **WinMain** необходимо организовать цикл обра-

ботки сообщений, поступающих в приложение. Работа этого цикла и тем самым приложения осуществляется до тех пор, пока пользователь не завершит приложение. При этом **WinMain** завершается и удаляется из списка активных задач.

В простейшем случае цикл обработки сообщений состоит из вызова функции **GetMessage**, и, если она возвращает ненулевое значение, вызывается функция **DispatchMessage**. Если в очереди потока появляется сообщение, то функция **GetMessage** забирает его из очереди и переносит в структуру **mess**.

В качестве параметров функции **GetMessage** указываются:

- адрес структуры **mess**, в которую поступает взятое из очереди сообщение,

- дескриптор окна, чьи сообщения будут обработаны **GetMessage**. Если **NULL**, то функция работает со всеми сообщениями данного приложения;

- два последних параметра определяют числовой диапазон сообщений, которые анализируются **GetMessage**. Для исключения фильтрации сообщений оба параметра должны быть равны нулю.

Особая ситуация возникает, когда **GetMessage** обнаруживает в очереди сообщение WM_QUIT с кодом **0x12**. В этом случае функция завершается с возвратом 0, что завершает цикл обработки сообщений.

Функция **DispatchMessage** вызывает оконную функцию окна того класса, которому предназначено сообщение и передает ей содержимое сообщения через структуру **mess**. После обработки сообщения оконной функцией управление возвращается в цикл обработки сообщений (рисунок 1.2).

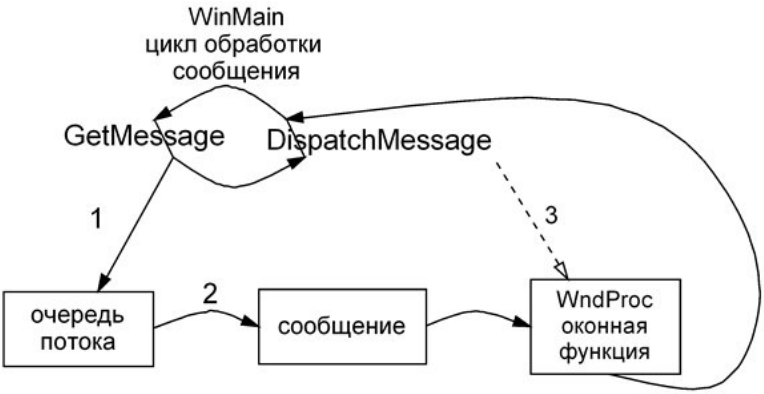


Рисунок 1.2 –Логика обработки сообщений в приложении

1.3 Оконная функция и способы передачи сообщений окнам

Оконная функция вызывается, когда в структуру **mess** попадает очередное сообщение, выбранное из входной очереди. Оконная функция должна проанализировать код сообщения и обработать его. С каждым окном связывается своя оконная функция. В программе на С необходимо описать прототипы все оконных функций, так как это прикладные функции с произвольными именами, а не системные, чьи прототипы описаны в WINUSER.H. Несмотря на то, что имена оконных функций произвольны, количество и типы ее параметров, а также тип возвращаемого значения в Windows жестко фиксированы. Описание оконной функции на С имеет вид:

```
LRESULT CALLBACK WndProc(HWND hwnd,UINT m,
                         WPARAM wParam,LPARAM lParam)
   { switch(mess)
       { case WM_DESTROY: // если пользователь
                          //завершил приложение
             PostQuitMessage(0);
             return 0; // возврат в Windows
         default:
     return(DefWindowProc(hwnd,mess,wParam,lParam));
  // остальные сообщения обрабатываем функцией
  // по умолчанию
       } // конец switch
   } // конец оконной функции
```

Описатель **CALLBACK** оконной функции эквивалентен описателю **STDCALL**, определяющему соглашение о связях вызывающей и вызываемой функций. В Win32 практически для всех функций действует соглашение, по которому параметры функции при ее вызове помещаются в стек в обратном порядке, т.е. в глубине стека помещается последний параметр, а на верхушке стека – первый. Извлекает их из стека сама функция, по прототипам описания заголовка функции легко вычислить количество байтов, занимаемых параметрами. Реальное освобождение памяти в стеке осуществляется командой **ret n**.

Рассмотрим структуру оконной функции. Она получает 4 параметра. Первый параметр **hwnd** представляет собой дескриптор окна, которому предназначено данное сообщение. Этот дескриптор нужен при обработке сообщений, например, в тех случаях, если на базе одного класса окна создается несколько окон, а поскольку оконная функция является общей для класса, то анализ дескриптора окна позволит иден-

тифицировать, в какое конкретно окно пришло сообщение.

Второй параметр **m** определяет код поступившего сообщения. Так как сообщений много, то по его коду определяется и выполняется конкретная ветвь оператора **switch** в теле оконной функции. Однако реально в программе нужно обрабатывать не все поступающие сообщения. Для того чтобы программист мог включить в оконную функцию обработку только своих сообщений, в Windows предусмотрена функция **DefWindowProc**, которая обрабатывает практически все сообщения, за исключением сообщения WM_DESTROY об уничтожении окна. Поэтому в простейшем случае в оконной функции необходимо реализовать обработку только этого сообщения. Рассмотрим, как это реализуется.

Пользователь может закрыть окно различными способами, однако во всех случаях Windows генерирует сообщение WM_DESTROY, которое направляется не в очередь потока, а сразу передается оконной функции. При обработке WM_DESTROY в оконной функции перед вызовом функции **PostQuitMessage** можно освободить созданные ресурсы, память, закрыть файлы и т.п. В простейшем случае можно вывести на экран предупреждающее сообщение. В минимальной оконной функции никаких действий не предусмотрено, сразу вызывается функция **PostQuitMessage**, генерирующая сообщение WM_QUIT, поступающее в очередь приложения. Это сообщение необходимо для того, чтобы функция **GetMessage** завершилась с false, прекратив тем самым цикл обработки сообщений.

Из вышеизложенного следует, что оконная функция может получить сообщение как из очереди потока, так и непосредственно. Реализуют эти механизмы передачи соответственно API-функции **PostMessage** и **SendMessage**.

Функция **PostMessage** помещает сообщение в очередь сообщений окна с указанным дескриптором и возвращает управление. Если сообщение помещено в очередь, то функция возвращает TRUE, иначе FALSE. Выбрано сообщение из очереди будет в цикле обработки сообщений.

Функция **SendMessage** осуществляет непосредственную передачу сообщения оконной функции. Отличие **SendMessage** от **PostMessage** состоит в том, что **SendMessage** вызывает оконную функцию и возвращает управление ***только после обработки сообщения.***

Если в приложении некоторые сообщения обрабатываются очень медленно (например, для обработки требуется один или несколько циклов), то во время этой обработки создается эффект зависания приложения. Для того чтобы во время долгой обработки проверить, есть

ли в очереди потока сообщение, используется функция **PeekMessage**, практически идентичная **GetMessage.**

PeekMessage имеет 5 параметров, первые 4 из которых совпадают с параметрами **GetMessage**:

- адрес структуры **mess**, в которую поступает взятое из очереди сообщение;
- дескриптор окна, чьи сообщения будут обработаны;
- минимальный и максимальный номера фильтруемых сообщений;
- флаг удаления сообщения, указывающий удалять ли сообщение из очереди после выборки (PM_NOREMOVE – оставить в очереди, PM_REMOVE – удалить из очереди).

Тогда при медленной обработке событий рекомендуется вставить фрагмент:

```
if (PeekMessage(&mess,NULL,0,0,PM_REMOVE))
      DispatchMessage(&mess);
```

1.4 Сообщения от клавиатуры

При нажатии и отпускании клавиш драйвер клавиатуры формирует сообщения и передает их в системную очередь сообщений. Затем сообщения от клавиатуры поступают в очередь сообщений приложения-владельца окна, имеющего фокус ввода (input focus).

Понятие фокуса ввода связано с активным окном. Окно, имеющее фокус ввода – это либо активное окно, либо дочернее окно активного окна. Окно является активным, если у него выделен заголовок, или рамка диалога, или текст заголовка в панели задач для минимизированного окна. Часто дочерними окнами для окна являются кнопки, переключатели и другие элементы управления, причем сами 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 дочернем окне, обычно показывается посредством мигающего курсора (для полей редактирования), рамки вокруг надписи на кнопке (для кнопок) или другими привлекающими внимание пользователя средствами. Если активное окно минимизировано, то окна с фокусом ввода нет.

Узнать о получении или потере фокуса ввода окном можно по информирующим сообщениям WM_SETFOCUS и WM_KILLFOCUS. Сообщение WM_SETFOCUS показывает, что окно получило фокус ввода, а WM_KILLFOCUS – что окно потеряло его. Определить или изменить окно, владеющее фокусом ввода, возможно с помощью функций **GetFocus** и **SetFocus**.

Типы сообщений, генерируемых драйвером клавиатуры. делятся на ***аппаратные*** и ***символьные***. Аппаратные сообщения связаны с нажатием и отпусканием клавиш. Каждой клавише на клавиатуре соответствуют два уникальных числа называемых *скан-кодом нажатия* и *скан-кодом отпускания* клавиши. Скан-код (*scan code)* является аппаратно-зависимым, то есть на клавиатурах разных ироизводителей одна и та же клавиша может иметь разные скан-коды. Когда пользователь нажимает или отпускает клавишу, драйвер клавиатуры считывает из порта 60h соответствующий скан-код, который преобразуется драйвером в *виртуальный код клавиши {virtual-key code).* Этот код является уже аппаратно-независимым, поэтому в Windows он однозначно идентифицирует нажатую или отпущенную клавишу. Завершающим действием драйвера является создание сообщения, содержащего скан-код, виртуальный код клавиши и другую информацию о нажатой/отпущенной клавише. Это аппаратное сообщение помещается в системную очередь сообщений. Далее Windows извлекает сообщение из системной очереди и передает его в очередь сообщений того потока, который является владельцем *активного* окна.

Виртуальные коды клавиш (virtual key code) определены в файле winuser.h и приведены в таблице 1.1.

Аппаратные сообщения бывают системные и несистемные. Несистемные аппаратные сообщения – это сообщения нажатия клавиши WM_KEYDOWN и ее отпускания WM_KEYUP. Если нажатие и отпускание клавиши необходимо Windows для своей обработки, то генерируются системные аппаратные сообщения WM_SYSKEYDOWN и WM_SYSKEYUP.

Эти сообщения обычно вырабатываются при нажатии клавиш в сочетании с клавишей <Alt> и служат для работы с меню или для системных функций, таких, например, как смена активного приложения (<Alt+Tab>).

Приложения обычно игнорируют сообщения WM_SYSKEYDOWN и WM_SYSKEYUP и передают их в функцию **DefWindowProc**, а оконная функция получает другие сообщения, являющиеся результатом обработки этих аппаратных сообщений клавиатуры (например, WM_COMMAND - выбор пункта меню).

Если в программе необходимо обрабатывать системные сообщения, то после их обработки следует вызвать **DefWindowProc**, чтобы Windows могла по-прежнему их использовать

Несистемные сообщения WM_KEYDOWN и WM_KEYUP обычно вырабатываются для клавиш, которые нажимаются и отпускаются без участия клавиши <Alt>. Приложение может использовать или не ис-

пользовать эти сообщения клавиатуры. Сама Windows их проигнорирует.

Таблица 1.1

Шестнадцатеричное значение	Идентификатор	Клавиатура IBM
03	VK_CANCEL	Ctrl + Break
08	VK_BACK	Backspace
09	VK_TAB	Tfab
0D	VK_RETURN	Enter
10	VK_SHIFT	Shift
11	VK_CONTROL	Ctrl
12	VK_MENU	Alt
13	VK_PAUSE	Pause
14	VK_CAPITAL	Caps Lock
1B	VK_ESCAPE	Esc
20	VK_SPACE	Пробел
21	VK_PRIOR	Page Up
22	VK_NEXT	Page Down
23	VK_END	End
24	VK_HOME	Home
25	VK_LEFT	Стрелка влево
26	VK_UP	Стрелка вверх
27	VK_RIGHT	Стрелка вправо
28	VK_DOWN	Стрелка вниз
2C	VK_SNAPSHOT	Print Screen
2D	VK_INSERT	Insert
2E	VK_DELETE	Delete
30...39		0...9 (на основной клавиатуре)
41...5A		A..Z
70...7B	VK_Fl...VK_F12	F1...F12
90	VK_NUMLOCK	Num Lock
91	VK_SCROLL	Scroll Lock

Обычно сообщения о нажатии и отпускании появляются парами. Однако если пользователь оставит клавишу нажатой так, чтобы включился автоповтор, то Windows посылает оконной процедуре серию сообщений WM_KEYDOWN (или WM_SYSKEYDOWN) и одно сооб-

щение WM_KEYUP (или WM_SYSKEYUP), когда, в конце концов, клавиша будет отпущена. Приложение с помощью функции **GetMessageTime** может узнать время нажатия и отпускания клавиши относительно старта системы.

Для всех ***аппаратных*** сообщений клавиатуры 32-разрядная переменная **lParam**, передаваемая в оконную процедуру, состоит из шести полей:

- счетчика повторений (число нажатий клавиши);
- скан-кода OEM (Original Equipment Manufacturer);
- флага расширенной клавиатуры (1, если сообщение клавиатуры появилось в результате работы с дополнительными клавишами расширенной клавиатуры IBM);
- кода контекста (1, если нажата клавиша <Alt>);
- флага предыдущего состояния клавиши (0, если в предыдущем состоянии клавиша была отпущена, и 1, если в предыдущем состоянии она была нажата),
- флага состояния клавиши (0, если клавиша нажимается, и 1, если клавиша отпускается).

Гораздо более важным параметром аппаратных сообщений клавиатуры, по сравнению с **lParam**, является параметр **wParam**. В этом параметре содержится виртуальный код клавиши, использующийся приложением для идентификации клавиши.

Параметры **wParam** и **lParam** аппаратных сообщений клавиатуры ничего не сообщают о состоянии так называемых клавиш сдвига (<Shift>, <Ctrl>, <Alt>) и клавиш-переключателей (<CapsLock>, <NumLock>, <ScrollLock>).

Приложение может получить текущее состояние (нажата или нет) любой виртуальной клавиши с помощью функций **GetKeyState** и **GetAsyncKeyState**.

Функция **GetKeyState** отражает состояние клавиатуры не в реальном времени, а только в момент, когда последнее сообщение от клавиатуры было выбрано из очереди. Эта функция не позволяет получать информацию о клавиатуре независимо от обычных сообщений от клавиатуры, сначала приложение должно получить сообщение от клавиатуры, а затем вызвать **GetKeyState** для определения состояния клавиш. Такая синхронизация дает преимущество, если нужно узнать положение переключателя для конкретного сообщения клавиатуры, даже если сообщение обрабатывается уже после того, как состояние переключателя было изменено.

Если действительно нужна информация о текущем положении клавиши, то можно использовать функцию **GetAsyncKeyState**.

Так как при обработке событий клавиатуры необходимо формировать символы, соответствующие нажатой клавише, то драйвер клавиатуры помимо аппаратных формирует ***символьные*** сообщения. Любая клавиша, например, <S>, в зависимости от состояния клавиш <Ctrl>, <Shift> и <CapsLock>, может использоваться для генерации разных символов - строчного ‘s’ или прописного ‘S’. Приложению посылаются оба аппаратных и символьное сообщение WM_CHAR.

Самостоятельное преобразования аппаратных сообщений клавиатуры в символьные сообщения в приложении не рекомендуется, так как необходимо знать об особенностях реализации каждой отдельной национальной клавиатуры. Это преобразование предлагается сделать самой Windows с помощью функции **TranslateMessage** в цикле обработки сообщений:

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

Функция **GetMessage** заполняет поля структуры mess данными следующего сообщения из очереди. Вызов **DispatchMessage** организует передачу управления оконной функции. Между двумя этими функциями находится вызов функции **TranslateMessage**, которая преобразует аппаратные сообщения в символьные. Если этим сообщением является WM_KEYDOWN (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.

Так же, как и для аппаратных сообщений, ***символьные*** сообщения бывают несистемные – WM_CHAR, WM_DEADCHAR и системные – WM_SYSCHAR, WM_SYSDEADCHAR.

Сообщения WM_SYSCHAR и WM_SYSDEADCHAR являются следствием сообщений WM_SYSKEYDOWN. В большинстве случаев можно обрабатывать только WM_CHAR.

Параметр **lParam** для WM_CHAR совпадает с **lParam** аппаратного сообщения клавиатуры, из которого сгенерировано символьное сообщение. Параметр **wParam** – это код символа ASCII.

Сообщения WM_DEADCHAR и WM_SYSDEADCHAR генерируются для неамериканских клавиатур с диакритическими знаками для

букв. Эти знаки называются немыми клавишами (dead keys), поскольку сами по себе клавиши с диакритическими знаками не определяют символ.

Рассмотрим, с какими наборами символов работает система Windows. Для обеспечения возможности работы с символами кириллицы фирма Microsoft разработала расширенный набор символов с кириллицей - набор символов OEM (Original Equipment Manufacturer). Сама же Windows для представления символов использует набор символов ANSI. В этом наборе определены не все коды и отсутствуют символы псевдографики.

По умолчанию для отображения символов выбирается системный шрифт, для которого используется набор символов ANSI.

1.5 Обработка сообщений от мыши

Определить наличие мыши в системе можно с помощью функции **GetSystemMetrics**, передав ей в качестве параметра значение SM_MOUSEPRESENT. Если мышь есть, эта функция возвращает ненулевое значение. Для определения количества кнопок мыши можно использовать вызов **GetSystemMetrics** с параметром SM_CMOUSEBUTTONS.

Когда пользователь перемещает мышь, Windows перемещает по экрану растровую картинку, которая называется курсор мыши (mouse cursor). Курсор мыши имеет вершину (hot spot) размером в один пиксель, точно указывающий положение мыши на экране.

Ресурсы Windows содержат несколько курсоров мыши, которые могут использоваться в приложениях. Наиболее типичным курсором является наклонная стрелка, которая называется IDC_ARROW (вершина курсора – острие стрелки). Курсор, в виде перекрестья (IDC_CROSS) имеет горячую точку в центре крестообразного шаблона. Курсор IDC_WAIT в виде песочных часов обычно используется для индикации того, что программа чем-то занята.

Кнопки трехкнопочной мыши обозначаются аббревиатурами LBUTTON, MBUTTON и RBUTTON.

Посылка сообщений от мыши и от клавиатуры различаются. Оконная функция получает сообщения мыши, когда мышь проходит через окно, при щелчке внутри окна, даже если окно не активно или не имеет фокуса ввода. Если мышь перемещается по клиентской области окна, то генерируется сообщение WM_MOUSEMOVE. Если кнопка мыши *нажимается* или *отпускается* внутри клиентской области, то оконная процедура получает следующие сообщения:

WM_LBUTTONDOWN, WM_MBUTTONDOWN, WM_RBUTTONDOWN – однократное нажатие левой, средней или правой кнопки, WM_LBUTTONUP, WM_MBUTTONUP, WM_RBUTTONUP – отпускание левой, средней или правой кнопки, WM_LBUTTONDBLCLK, WM_MBUTTONDBLCLK, WM_RBUTTONDBLCLK – двойной щелчок левой, средней или правой кнопки.

Для всех этих сообщений значение параметра **lParam** содержит положение мыши, причем в младшем слове находится значение координаты X, а в старшем слове — значение координаты Y от левого верхнего угла клиентской области окна. Эти значения можно извлечь из **lParam** при помощи макросов LOWORD и HIWORD. Значение параметра **wParam** показывает состояние кнопок мыши и клавиш Shift и Ctrl. Можно проверить параметр wParam с помощью битовых масок, определенных в заголовочных файлах:

MK_LBUTTON – левая кнопка нажата.

MK_RBUTTON – правая кнопка нажата.

MK_MBUTTON – средняя кнопка нажата.

MK_SHIFT – клавиша Shift нажата.

MK_CONTROL – клавиша Ctrl нажата.

Состояние кнопок мыши или клавиш Shift и Ctrl можно получить также с помощью функции **GetKeyState**.

При обработке двойного щелчка мыши следует отметить, что окно будет получать сообщения о двойном щелчке только если стиль соответствующего класса окна содержит флаг CS_DBLCLKS. Поэтому перед регистрацией класса окна нужно присвоить полю **style** структуры WNDCLASS значение, включающее этот флаг.

Для мыши с колесом при нажатии на колесо Windows генерирует такие же сообщения, какие вырабатываются при нажатии средней кнопки трехкнопочной мыши. Прокрутка колеса приводит к генерации сообщения WM_MOUSEWHEEL. Если нужно обработать прокрутку колеса, то рекомендуется подключить файл ***zmouse.h.*** Младшая часть параметра **wParam** сообщения WM_MOUSEWHEEL показывает состояние кнопок и клавиш Shift и Ctrl. Старшая часть **wParam** содержит значение, отображающее расстояние, пройденное колесом. Оно рассчитывается как количество шагов колеса при прокрутке, умноженное на коэффициент WHEEL_DELTA. В файле zmouse.h этот коэффициент равен 120.

В Windows для мыши определен набор из 21 сообщения. Однако 11 из этих сообщений не относятся к клиентской области, и программы для Windows обычно игнорируют их.

Количество посылок сообщения WM_MOUSEMOVE зависит от устройства мыши и скорости, с которой оконная функция может обра-

батывать сообщения о движении мыши. Если мышь оказывается вне клиентской области окна, но все еще внутри окна, то Windows посылает оконной процедуре сообщения мыши, связанные с неклиентской областью, включающей заголовок, меню, рамку и полосы прокрутки.

Сообщения мыши для неклиентской области содержат в названиях буквы **NC**, что означает неклиентская (nonclient). Например, если мышь перемещается внутри неклиентской области окна, то оконная функция получает сообщение WM_NCMOUSEMOVE. Однако параметры **wParam** и **lParam** для таких сообщений, связанных с неклиентской областью, имеют другой смысл: Параметр **wParam** показывает зону неклиентской области, в которой произошло перемещение или щелчок. Его значение совпадает с одним из идентификаторов, начинающихся с HT, что означает тест попадания (hit-test). Сообщения теста попадания будут рассмотрены ниже.

Параметр **lParam** содержит в младшем слове значение координаты X, а в старшем – Y. Однако эти координаты являются координатами относительно начала экрана, а не относительно начала клиентской области. Значения координат X и Y верхнего левого угла экрана равны 0.

Приложение может преобразовать экранные координаты в координаты клиентской области окна и наоборот с помощью функций **Windows ScreenToClient** и **ClientToScreen**.

Сообщение WM_NCHITTEST (тест попадания в нерабочую область – nonclient hit-test) предшествует всем остальным сообщениям мыши клиентской и неклиентской области. Параметр **lParam** содержит значения X и Y экранных координат положения мыши. Параметр wParam не используется.

В приложениях Windows это сообщение обычно передается в **DefWindowProc**. В этом случае Windows использует сообщение WM_NCHITTEST для выработки всех остальных сообщений на основе положения мыши.

Для сообщений мыши неклиентской области возвращаемое значение функции **DefWindowProc** при обработке сообщения WM_NCHITTEST передается как параметр **wParam** в сообщении мыши.

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

Рассмотренные выше сообщения приложение получает в случаях, когда курсор мыши находится в клиентской или в неклиентской области окна. Но иногда приложению может понадобиться получать со-

общения от мыши даже в случаях, когда курсор мыши находится вне окна. Если это необходимо сделать, то приложение может произвести захват (capture) мыши.

Захват мыши осуществляется с помощью функции **SetCapture**. После вызова этой функции, Windows посылает все сообщения мыши в оконную функцию того окна, чей дескриптор окна был передан в функцию **SetCapture**. Пока мышь захвачена, системные функции клавиатуры тоже не действуют.

Сообщения мыши в захваченном состоянии всегда остаются сообщениями клиентской области, даже если мышь оказывается не в клиентской области окна. Параметр **lParam,** однако, показывает положение мыши в координатах клиентской области. Эти координаты, однако, могут стать отрицательными, если мышь окажется левее или выше клиентской области.

Освободить мышь можно при помощи функции **ReleaseCapture**.

При обработке сообщений от мыши следует отметить, что если системное модальное окно сообщений или системное модальное окно диалога находится на экране, никакая другая программа не может получать сообщения от мыши.

1.6 Использование таймера в приложениях

Для отслеживания временных промежутков в системе или выполнения действий в программе с требуемой периодичностью в Win32 реализован ряд API-функций. При работе с ними необходимо помнить, что Windows не является ОС реального времени, и точность подсчета времени зависит от возникновения прерываний в системе и количества одновременно запущенных приложений. Под *временем Windows* понимается количество миллисекунд, прошедших с момента старта ОС. Это время увеличивается на период системного таймера, и дополнительно синхронизируется с часами реального времени RTC (Real Time Clock).

Для получения текущего значения времени Windows используется функция **GetTickCount**, возвращающая число миллисекунд с момента старта ОС. Параметров функция не имеет. **GetTickCount** очень удобно использовать для измерения времени выполнения фрагмента программы, вызвав ее 2 раза - до начала фрагмента и после фрагмента:

```
t1=GetTickCount();
// фрагмент кода
t2= GetTickCount();
t3=t2-t1;
```

Наряду с временем Windows существует понятие системного времени – это текущее время по Гринвичу (часы, минуты, секунды и миллисекунды) и дата (день недели, число, месяц, год). Для получения системного времени используется функция **GetSystemTime**, параметром которой является указатель на структуру типа SYSTEMTIME. Для получения времени, отражаемого на компьютере, используется **GetLocalTime**.

Чтобы выполнить некоторый код в программе с заданной периодичностью, используется стандартный таймер, инициализируемый функцией **SetTimer**. Параметрами этой функции являются:

- дескриптор окна, связанного с таймером. Если дескриптор не указан, то сообщения от таймера посылаются в его собственную функцию обработки сообщений;
- идентификатор таймера (целое число, большее нуля), если предполагается использовать несколько таймеров;
- интервал от 1 до 4294967295 мс, что соответствует 50 дням. Интервал указывает периодичность посылки сообщения от таймера WM_TIMER либо оконной функции окна с указанным дескриптором, либо собственной функции таймера;
- адрес функции обработки сообщений таймера или NULL, если обработка происходит в оконной функции. Собственная функция обработки сообщений от таймера должна быть объявлена с атрибутом CALLBACK.

Для уничтожения таймера, после того как в нем исчезла надобность, используется функция **KillTimer**, которой передаются дескриптор окна и идентификатор таймера, который надо уничтожить.

Отметим, что если сообщение WM_TIMER не успевает обработаться с нужной периодичностью, то в очереди эти сообшения не накапливаются, а объединяются в одно сообщение WM_TIMER, и потеря сообщений никак не фиксируется в системе. Это может приводить к существенному разбросу реального интервала срабатывания таймера.

1.7 Вызов функций WinAPI в среде MASM32

MASM32 использует набор макросов, позволяющий приблизить синтаксис ассемблера к синтаксису языков высокого уровня и уменьшить количество ошибок. Каждый макрос имеет имя и формальные параметры, однако, в отличие от классических макросредств ассемблера, имена формальных параметров могут быть снабжены дополнительными модификаторами. Например, рассмотрим макрос с именем Prim, который имеет 4 параметра, из них первый обязателен, второй по

умолчанию инициализируется заданным значением, третий описан стандартным образом, а вместо четвертого в тело макроса может быть подставлена строка, указанная в макровызове:

```
Prim MACRO p1:REQ, p2:= <eax>,p3, p4:VARARG
```

Ключевое слово REQ указывает, что параметр является обязательным. Если при макровызове второй параметр не будет указан, то он будет равен значению по умолчанию. Модификатор **VARARG** указывается только для последнего параметра и означает, что этот параметр при макроподстановке заменяется на оставшуюся строку фактических параметров, включая запятые между ними. Для данного примера можно указать больше 4 параметров, и тогда весь текст начиная с 4 параметра ассоциируется с формальным параметром p4, вместо которого подставится в тело макроса.

Рассмотрим примеры макросов. Для API-функций в качестве аргументов часто бывает необходимо описать строку символов, заканчивающуюся нулем. Это делает макрос:

```
szText MACRO Name, Text:VARARG
 LOCAL lbl
   jmp lbl
     Name db Text,0
   lbl:
 ENDM
```

Для возврата из функции результата ее работы через ***eax*** используется макрос:

```
return MACRO arg
 mov eax, arg
 ret
ENDM
```

В MASM присутствует поддержка высокоуровневых операторов передачи управления. Они часто используются для облегчения восприятия исходного кода. Макрос **.IF** с директивами **.ELSE**, **.ELSEIF**, **.ENDIF** используются для организации ветвлений, директивы **.WHILE/ .ENDW** и **.REPEAT/ .UNTIL [CXZ]** позволяют организовать циклы. **.BREAK** производит выход из цикла, а **.CONTINUE** - переход на новую итерацию.

Удобство условного и циклического макросов заключается в возможности использования в выражениях знаков сравнений и логиче-

ских операций, связывающих мнемоники регистров и имен переменных.

Использование макроса **.IF** позволяет организовать перебор значений в оконной функции, аналогичный оператору switch. Он имеет синтаксис:

```
.IF условие1
       операторы 1
 [.ELSEIF] условие2
              операторы2
                ...
 [.ELSE]
    операторы N
 .ENDIF
```

Этот макрос генерирует сравнение и условные переходы таким образом, чтобы при истинности **условия1** выполнялись **операторы1** до тех пор, пока не встретится следующий .**ELSEIF, .ELSE**, или **.ENDIF**. Макросы **.IF** могут быть вложенными. Оператор **.ELSEIF** используется для выделения блока операторов, выполняющихся, если предшествующие условия в **.IF** и **.ELSEIF** ложны, а условие в текущем **.ELSEIF** истинно. Внутри **.IF** могут быть несколько блоков **.ELSEIF**. **.ELSE** определяет выполняемый блок операторов, если все предыдущие условия были ложны. **.ENDIF** закрывает текущий **.IF.**

Макрос **REPEAT** выполняет тело макроса, пока условие не станет истинным, например, следующий макрос повторяет код между repeat и until, пока значение регистра ebx не станет равным 5:

```
.REPEAT
; тело макроса
.UNTIL ebx==5
```

Макрос **WHILE** служит для организации выполнения тела макроса до тех пор, пока условие истинно:

```
.WHILE ebx==5
; тело макроса
.ENDW
```

Чтобы прервать цикл и выйти из него, используется директива .BREAK:

```
mov eax,2
```

```
.WHILE ebx==5
inc eax
.IF eax==4
.BREAK
.ENDIF
.ENDW
```

Если eax=4, цикл будет прерван.

В макросах .**REPEAT** и .**WHILE** для перехода на проверку условия цикла используется директива **CONTINUE**.

Для упрощения вызова API-функций используется макрос **invoke**. Пусть необходимо вызвать API-функцию с именем **FunctionName**, имеющую 4 параметра. Четвертый и второй параметры представляют собой адреса некоторых объектов. Традиционный вызов функции имеет вид:

```
push offset par4
push par3
push offset par2
push par1
call FunctionName
```

С использованием **invoke** вызов имеет вид:

```
invoke FunctionName,par1,ADDR par2,par3,ADDR par4
```

Рассмотрим программу на MASM32, выводящую окно сообщения с кнопкой OK, текстом и заголовком.

```
.model flat, stdcall
   option casemap :none   ; case sensitive
   include \masm32\include\windows.inc
   include \masm32\include\user32.inc
   include \masm32\include\kernel32.inc
   szText MACRO Name, Text:VARARG
   LOCAL lbl
       jmp lbl
          Name db Text,0
       lbl:
   ENDM
   includelib \masm32\lib\user32.lib
   includelib \masm32\lib\kernel32.lib
.code
start:
```

```
    szText szDlgTitle,"First MASM"
    szText szMsg,"Assembler "
    invoke MessageBox,0,ADDR szMsg,ADDR szDlgTitle,
          MB_OK
    invoke ExitProcess,0
end start
```

Используем плоскую модель памяти и стандартный вызов подпрограмм, согласно которому параметры помещаются в стек в порядке справа налево, а убираются из стека подпрограммой за счет **ret N**. Для работы программы в нее необходимо включить файл **windows.inc**, содержащий описание всех констант, макроопределений и структур Windows, используемых в системе. Например, определение структуры сообщения на Ассемблере имеет вид:

```
MSG STRUCT
  hwnd      DWORD       ?
  message   DWORD       ?
  wParam    DWORD       ?
  lParam    DWORD       ?
  time      DWORD       ?
  pt        POINT       <>
MSG ENDS
```

Определение структуры для регистрации класса окна **WNDCLASS**:

```
WNDCLASS STRUCT
  style             DWORD       ?
  lpfnWndProc       DWORD       ?
  cbClsExtra        DWORD       ?
  cbWndExtra        DWORD       ?
  hInstance         DWORD       ?
  hIcon             DWORD       ?
  hCursor           DWORD       ?
  hbrBackground     DWORD       ?
  lpszMenuName      DWORD       ?
  lpszClassName     DWORD       ?
WNDCLASS ENDS
```

Если программисту не нужны все описания из **windows.inc**, он должен явно включить в программу только нужные описания системных констант и структур данных.

Включаемые в программу **kernel32.inc** и **user32.inc** содержат описания прототипов всех функций, содержащихся соответственно в используемых библиотеках функций **kernel32.lib** и **user32.lib**. Прототи-

пы нужны для облегчения проверки при трансляции количества и размеров параметров процедуры. Описания прототипов имеют вид:

```
Имя_процедуры PROTO список_описаний_типов
```

Список_описаний_типов состоит из разделенных запятыми конструкций, начинающихся двоеточием, за которым следует ключевое слово, характеризующее размер параметра – DWORD, WORD, BYTE. Например, прототипы рассмотренных **WinMain** и **WndProc** имеют вид:

```
WinMain PROTO :DWORD,:DWORD,:DWORD,:DWORD
WndProc PROTO :DWORD,:DWORD,:DWORD,:DWORD
```

Если программисту нужны не все функции из библиотек, а только некоторые, он должен явно описать прототипы используемых внешних процедур и подключить только библиотеки без файлов с расширением **inc**.

С учетом вышеизложенного, описание функции WinMain на MASM32 имеет вид:

```
hWnd  dd  ?
WinMain proc hInst:DWORD, hPrevInst :DWORD,
             CmdLine:DWORD, CmdShow :DWORD
  LOCAL wc:WNDCLASSEX
  LOCAL msg:MSG
  szText szClassName,"My_Class"
  szText szAppName,"Application"
; заполнение полей wc
  mov wc.lpfnWndProc, offset WndProc
;адрес  WndProc
  push hInst
  pop wc.hInstance
  mov wc.lpszClassName, offset szClassName
  invoke LoadIcon,hInst,500
  mov wc.hIcon, eax
  invoke LoadCursor,NULL,IDC_ARROW
  mov wc.hCursor,eax
  invoke RegisterClassEx,ADDR wc
; регистрация класса окна
  invoke CreateWindowEx, NULL, addr szClassName,
       addr szAppName,  WS_OVERLAPPEDWINDOW,
       100,100, 400, 200, NULL, NULL, hInst, NULL
  mov hWnd, eax
```

```
   invoke ShowWindow, hWnd, SW_SHOWNORMAL
   invoke UpdateWindow, hWnd
   .while TRUE
      invoke GetMessage, addr msg, NULL, 0, 0
   .break .if (!eax)
      invoke     TranslateMessage, addr msg
      invoke     DispatchMessage, addr msg
   .endw
   mov eax, msg.wParam
   ret
WinMain endp
```

Оконная функция на MASM32:

```
WndProc proc hWin:DWORD, uMsg:DWORD, wParam:DWORD,
             lParam :DWORD
   .if uMsg == WM_CLOSE
      szText TheText,"Please Confirm Exit"
      invoke MessageBox,hWin,ADDR TheText,
             ADDR szDisplayName, MB_YESNO
      .if eax == IDNO
              return 0
       .endif
    .elseif uMsg == WM_DESTROY
        invoke PostQuitMessage,NULL
        return 0
     .endif
    invoke DefWindowProc,hWin,uMsg,wParam,lParam
    ret
WndProc endp
```

Процесс компиляции программ на MASM32 осуществляет Ml.exe, а редактированием связей занимается Link.exe.

1.8 Примеры программ по обработке сообщений от клавиатуры и мыши

Пример 1. Реализовать изменение текста заголовка окна на заданный текст по двойному щелчку правой кнопки в клиентской области окна Обратную замену заголовка осуществить по нажатию клавиши F1.

```
.386
.model flat,stdcall
option casemap:none
```

```
include c:\masm32\include\windows.inc
include c:\masm32\include\user32.inc
include c:\masm32\include\kernel32.inc
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib
WinMain proto :DWORD,:DWORD,:DWORD,:DWORD
.data
AppName  db "Мое приложение",0
NewName db "Новое название!!!",0
hInstance  HINSTANCE ?
CommandLine LPSTR ?
.code
start:
  invoke GetModuleHandle, NULL
  mov    hInstance,eax
  invoke GetCommandLine
  mov    CommandLine,eax
  invoke WinMain, hInstance,NULL,CommandLine,
         SW_SHOWDEFAULT
;запуск WinMain и после ее завершения остановка
;процесса
  invoke ExitProcess,eax
WinMain proc hInst:HINSTANCE,hPrevInst:HINSTANCE,
             CmdLine:LPSTR,CmdShow:DWORD
  LOCAL wc:WNDCLASSEX
; WNDCLASSEX аналогична WNDCLASS за исключением 2
;полей cbSize – размера структуры в байтах и hIconSm –
;дескриптора пиктограммы приложения размером 16x16,
;используемой в заголовке окна данного класса
  LOCAL msg:MSG
  LOCAL hwnd:HWND
  LOCAL X:DWORD
  LOCAL Y:DWORD
  mov X,500
  mov Y,350
  mov   wc.cbSize,SIZEOF WNDCLASSEX
  mov   wc.style, CS_HREDRAW or CS_VREDRAW OR
                  CS_DBLCLKS
; стиль класса обеспечивает реакцию на двойные щелчки
  mov   wc.lpfnWndProc, OFFSET WndProc
  mov   wc.cbClsExtra,NULL
  mov   wc.cbWndExtra,NULL
  push  hInstance
  pop   wc.hInstance
  mov   wc.hbrBackground,COLOR_BTNFACE+1
  mov   wc.lpszMenuName,NULL
  mov   wc.lpszClassName,OFFSET AppName
```

```
  invoke LoadIcon,NULL,IDI_APPLICATION
  mov   wc.hIcon,eax
  mov   wc.hIconSm,eax
  invoke LoadCursor,NULL,IDC_ARROW
  mov   wc.hCursor,eax
  invoke RegisterClassEx, addr wc
  INVOKE CreateWindowEx,NULL,ADDR AppName,
         ADDR AppName, WS_OVERLAPPEDWINDOW,200, 200,
         X,Y,NULL, NULL,hInst,NULL
  mov   hwnd,eax
  invoke ShowWindow, hwnd,SW_SHOWNORMAL
  invoke UpdateWindow, hwnd
  .WHILE TRUE
  begin:  invoke GetMessage, ADDR msg,NULL,0,0
    .BREAK .IF (!eax)
    invoke TranslateMessage, ADDR msg
    invoke DispatchMessage, ADDR msg
  .ENDW
  mov eax,msg.wParam
  ret
WinMain endp

WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM,
             lParam:LPARAM
  .IF uMsg==WM_DESTROY
    invoke PostQuitMessage,NULL
  .ELSEIF uMsg==WM_RBUTTONDBLCLK
;заменяем текст заголовка окна
     invoke SetWindowText,hWnd,ADDR NewName
  .ELSEIF uMsg==WM_KEYDOWN
       ;восстанавливаем текст
    .IF wParam == VK_F1
      invoke SetWindowText,hWnd,ADDR AppName
    .endif
    .ELSE
      invoke DefWindowProc,hWnd,uMsg,wParam,lParam
      ret
    .ENDIF
  xor eax,eax
  ret
WndProc endp
end start
```

Пример 2. Двойной щелчок левой кнопки в клиентской области окна приводит к циклическому перемещению окна по экрану. Отменить эти действия можно нажатием клавиши на клавиатуре.

Рассмотрим только оконную функцию и макросы, осуществляющие сложение и вычитание параметров, возвращающие результат операции через первый операнд:

```
plus MACRO mem,vl      ; mem=mem+vl
       push eax
       mov eax,mem
       add eax,vl
       mov mem,eax
       pop eax
ENDM
minus MACRO mem,vl     ; mem=mem-vl
       push eax
       mov eax,mem
       sub eax,vl
       mov mem,eax
       pop eax
ENDM
```

Так как перемещение окна должно быть циклическим, то целесообразно использовать сообщение от таймера WM_TIMER, при обработке которого осуществлять перемещение окна в новую позицию. Создание таймера будем осуществлять при обработке сообщения о двойном щелчке левой кнопки мыши. Уничтожение таймера реализуем при обработке сообщения о нажатии любой клавиши.

```
WndProc proc hWin   :DWORD,
             uMsg   :DWORD,
             wParam :DWORD,
             lParam :DWORD
; начальные значения координат левого верхнего угла
; (Left,Top) окна задаются как глобальные переменные
       .if uMsg == WM_TIMER
              .if Left == 50
                     .if Top < 350
                            plus Top,10
                     .else
                            plus Left,10
                     .endif
              .endif
              .if Top == 350
                     .if Left < 350
                            plus Left,10
                     .else
```

```
                        minus Top,10
                .endif
        .endif
        .if Left == 350
                .if Top > 50
                        minus Top,10
                .else
                        minus Left,10
                .endif
        .endif
        .if Top == 50
                .if Left > 50
                        minus Left,10
                .else
                        plus Top,10
                .endif
        .endif
; вывод окна с измененными координатами левого
; верхнего угла
        invoke SetWindowPos,hWnd, HWND_TOP,
               Left,Top,200,100,SWP_SHOWWINDOW
    .elseif uMsg == WM_KEYDOWN
; при нажатии любой клавиши таймер уничтожается
        invoke KillTimer, hWin, NULL
    .else
    .if uMsg == WM_LBUTTONDBLCLK
; инициализируем таймер
        invoke SetTimer, hWin, NULL, 500, NULL
; 500 мс = 0,5 секунды
    .else
        .if uMsg == WM_CLOSE
        .elseif uMsg == WM_DESTROY
               invoke PostQuitMessage,NULL
               return 0
        .endif
    .endif
    .endif
    invoke DefWindowProc,hWin,uMsg,wParam,lParam
    ret
WndProc endp
```

1.9 Упражнения

С использованием API-функций **AnimateWindow, CloseWindow, FindWindow, FlashWindow, FlashWindowEx, GetClientRect, GetParent, GetDesktopWindow, GetTitleBarInfo, GetWindowPlacement**, **IsChild, GetWin-**

dowTextLength, IsWindow, IsWindowVisible, MoveWindow, CloseWindow, OpenIcon, SetWindowPlacement, SetWindowText, ShowWindow, WindowFromPoint реализовать предложенные задания.

1. После двойного щелчка левой кнопки в клиентской области окна окно начинает перемещаться по вертикали. Нажатие любой клавиши прекращает движение.
2. Щелчок правой кнопкой на клиентской области окна приводит к изменению его размеров (уменьшению в 2 раза). Нажатие любой клавиши возвращает исходные размеры.
3. Щелчок правой кнопкой на клиентской области окна приводит к изменению реакции на нажатие цифр. Каждое следующее нажатие цифры изменяет размер окна. Нажатие любой другой клавиши возвращает исходные размеры.
4. Щелчок правой кнопкой в верхней половине окна приводит к его свертке. Развертка окна – по нажатию любой функциональной клавиши.
5. Перемещение мыши в верхней половине окна приводит к появлению периодического мерцания заголовка окна. Нажатие любой буквы из линейки «QWER..» перемещает окно в правый верхний угол экрана.
6. Создать два окна, одно из которых – родительское для другого. Перемещение мыши в неклиентской области дочернего окна приводит к его свертке. Нажатие клавиши «пробел» при установке фокуса ввода на родительское окна приводит также к его свертке.
7. Щелчок левой кнопкой в неклиентской области окна перемещает окно вправо на заданное количество пикселов. Нажатие клавиши Alt вместе с цифровой клавишей возвращает окно на место.
8. Щелчок правой кнопкой в неклиентской области окна приводит к тому, что последующее нажатие любой буквенной клавиши сдвигает окно на заданное число пикселов вправо.
9. Двойной щелчок левой кнопки в рабочей области окна приводит к тому, что при нажатии цифровых клавиш окно сдвигается вниз на количество пикселов, соответствующее нажатой цифре.
10. Двойной щелчок правой кнопки в клиентской области окна приводит к тому, что при нажатии клавиш стрелок впоследствии реализуется перемещение окна по экрану.
11. Щелчок правой кнопкой в нижней половине окна разрешает изменение текста в заголовке окна. Дальнейшее нажатие букв приводит к изменению текста заголовка окна на нажатую букву.
12. Перемещение мыши в нижней половине окна максимизирует окно. Нажатие ESC – возврат окна к прежнему виду.

13. Двойной щелчок левой кнопкой в неклиентской области окна максимизирует окно. Нажатие любой буквы – возврат к прежним размерам.

14. Двойной щелчок правой кнопкой в неклиентской области окна приводит к его перемещению в цикле из его исходного состояния в горизонтальном направлении влево и вправо до тех пор, пока не будет нажата любая клавиша из нижней линейки.

15. Одновременное нажатие кнопок в неклиентской области окна приводит к его перемещению по экрану в вертикальном направлении после нажатия клавиш из линейки «ASD..».

16. Перемещение мыши в неклиентской области окна приводит к изменению его размеров. Нажатие клавиши ESC запрещает подобное изменение.

17. Двойной щелчок правой кнопки в клиентской области окна изменяет текст заголовка окна на заданный текст. Обратная замена – клавиша F1.

18. Создать два окна, одно из которых частично перекрывает другой. Щелчок правой кнопкой в левой половине верхнего окна активизирует перекрытое окно. Если до щелчка нажать клавишу «пробел», то перекрытое окно не активизируется.

19. Перемещение мыши в левой половине окна изменяет цвет фона окна (**SetClassLongPtr**). Нажатие клавиши F1 отменяет изменение.

20. Двойной щелчок левой кнопкой в неклиентской области окна свертывает его в пиктограмму. Нажатие любой функциональной клавиши разворачивает окно.

21. Создать три окна, одно из них показать. Двойной щелчок правой кнопкой в неклиентской области окна приводит к выводу на экран остальных окон, если они еще не показывались. Вывод созданных, но не показанных окон разрешен только в случае, если не нажата клавиша Enter.

22. Перемещение мыши в неклиентской области окна приводит к перемещению окна в левый верхний угол экрана. Нажатие клавиш PgUp и PgDn приводит к перемещению окна из левого верхнего угла в левый нижний угол и обратно.

23. Щелчок правой кнопкой в неклиентской области окна отменяет реакцию на нажатие клавиш End и Home, которые перемещают выведенное окно соответственно между левым верхним и правым нижним углом экрана.

24. Двойной щелчок левой кнопки в клиентской области окна приводит к смене заголовка окна. Выбор текущего заголовка окна зависит от количества нажатий клавиши Insert. Каждое следующее на-

жатие делает активным следующий по прядку заголовок из некоторого массива заголовков (циклически).

25. Вывести на экран два окна. Двойной щелчок правой кнопки в клиентской области любого из окон меняет их местами. Обратный обмен - по нажатию клавиши Delete.

26. Вывести на экран два окна. Щелчок правой кнопкой в правой половине второго окна изменяет заголовок первого. Обратный обмен – по нажатию клавиши Insert.

27. Вывести на экран три окна. Щелчок левой кнопкой в правой половине любого окна приводит к свертке в пиктограмму остальных. Обратное разворачивание, если они уже свернуты, осуществить при нажатии любой цифровой клавиши.

28. Двойной щелчок левой кнопкой в неклиентской области окна приводит к выводу на экран дочернего окна. Свернуть в пиктограмму оба окна можно при нажатии клавиш F1-F5.

29. Создать два окна, одно из них показать. Двойной щелчок правой кнопкой в неклиентской области показанного окна приводит сначала к показу второго окна, потом к его сворачиванию, потом к его разворачиванию, потом к его закрытию.

30. Создать два окна. Щелчок левой кнопкой в неклиентской области любого окна приводит к перемещению их в левый верхний угол экрана так, чтобы меньшее было сверху. Нажатие любой гласной буквы закрывает оба окна.

31. Создать два окна. Перемещение мыши в неклиентской области первого окна вызывает перемещение второго окна по горизонтали от левого к правому краю экрана и обратно. Нажатие любой цифры закрывает оба окна.

32. Двойной щелчок левой кнопки в клиентской области окна приводит к выводу на экран двух дочерних окон. По нажатию клавиши Esc заголовки дочерних окон меняются местами.

33. Вывести на экран два окна. Двойной щелчок правой кнопки в клиентской области первого окна приводит к последовательной свертке и развертке второго окна до тех пор, пока не нажата клавиша «пробел».

34. Отпускание левой кнопки мыши в верхнем левом квадранте окна изменяет текст заголовка окна на одну из нескольких строк текста в зависимости от координаты отпускания. Этот режим активен только после нажатия одной из клавиш-стрелок.

35. Щелчок левой кнопкой в верхнем левом квадранте окна приводит к его перемещению на столько пикселов влево, каков порядко-

вый номер щелчка. Максимизация окна – нажатие клавиши «стрелка вверх».

36. Двойной щелчок левой кнопкой в неклиентской области окна приводит к созданию другого окна, цвет фона которого зависит от координаты щелчка. Нажатие любой цифры закрывает созданное окно.

37. Вывести окно в правой половине экрана. Двойной щелчок правой кнопкой в неклиентской области окна приводит к тому, что каждое последующее нажатие клавиши «пробел» сдвигает окно влево.

38. Щелчок левой кнопкой в области заголовка окна изменяет этот заголовок и выводит на экран второе окно. Закрытие обоих окон происходит при нажатии любой функциональной клавиши.

39. Нажатие клавиши Insert запрещает реакцию на щелчок правой кнопкой в неклиентской области окна. Реакция на такой щелчок заключается в перемещении окна по экрану из левого нижнего в правый верхний угол.

40. Одновременный щелчок кнопок в клиентской области окна приводит к выводу на экран дочернего окна. По нажатию клавиши Home оба эти окна перемещаются к правой границе экрана.

41. Вывести на экран три окна с различными цветами фона. Двойной щелчок правой кнопки в клиентской области любого окна сворачивает в пиктограмму остальные окна. Их разворачивание – по нажатию цифр на дополнительной клавиатуре.

42. Щелчок правой кнопкой в верхнем правом квадранте окна выводит новое окно, цвет фона которого зависит от места щелчка (предусмотреть не менее 3 вариантов). Нажатие функциональной клавиши закрывает созданные окна.

43. Вывести на экран три окна, одно из которых – родительское для двух других. Щелчок левой кнопкой в верхнем правом квадранте родительского окна закрывает дочерние окна. Нажатие любой буквенной клавиши закрывает и родительское окно.

44. Вывести на экран три окна. Одновременный щелчок кнопок в неклиентской области любого окна закрывает два из них, и третье окно получает возможность перемещаться к левой границе экрана при нажатии клавиши End.

45. Двойной щелчок правой кнопкой в неклиентской области окна разрешает следующую реакцию на нажатие цифр на дополнительной клавиатуре. Каждое нажатие цифры уменьшает ширину окна на столько пикселов, какова цифра.

46. Щелчок левой кнопкой в неклиентской области окна приводит к изменению заголовка окна на новую строку в зависимости от пред-

варительного нажатия буквенных клавиш. По номеру линейки выбирается новый заголовок.

47. Щелчок правой кнопкой в неклиентской области окна вызывает его циклическое перемещение по экрану. Эта реакция на кнопку разрешается при предварительном нажатии не менее 5 буквенных клавиш.

48. Двойной щелчок левой кнопки в клиентской области окна вызывает смену заголовка окна на букву, клавиша которой была нажата перед щелчком.

49. Двойной щелчок правой кнопки в клиентской области окна выводит на экран дочернее окно, цвет фона которого зависит от координаты щелчка. Реализовать не менее 4 вариантов.

50. Щелчок правой кнопкой в нижнем левом квадранте окна сворачивает его в пиктограмму, а нажатие клавиши F8 разворачивает окно при условии, что перед этим была нажата хотя бы одна цифра.

51. Щелчок левой кнопкой в нижнем левом квадранте окна разрешает перемещение окна по экрану с помощью клавиш-стрелок, PgUp, PgDn.

52. Двойной щелчок левой кнопкой в неклиентской области окна позволяет в дальнейшем при нажатии клавиш 1-4 двигать окно по экрану в четырех направлениях.

53. Двойной щелчок правой кнопкой в неклиентской области окна выводит на экран два дочерних окна с разными стилями. Нажатие любой функциональной клавиши закрывает одно из дочерних окон.

54. Щелчок левой кнопкой в неклиентской области окна выводит длину заголовка окна. Нажатие любой буквы меняет заголовок на эту букву.

55. Щелчок правой кнопкой в неклиентской области окна разрешает перемещение окна в левый верхний угол (по нажатии F1), левый нижний угол (F2), правый верхний (F3), правый нижний (F4).

56. Двойной щелчок левой кнопки в клиентской области окна разрешает создание дочернего окна по нажатию цифры. Заголовком дочернего окна должна стать эта цифра.

57. Двойной щелчок правой кнопки в клиентской области окна закрывает все дочерние окна, созданные по нажатии клавиш F1-F4.

58. Щелчок правой кнопкой в нижнем правом квадранте окна разрешает перемещение окна при нажатии стрелок на дополнительной клавиатуре.

59. Щелчок левой кнопкой в нижнем правом квадранте окна приводит к созданию дочернего окна, заголовок которого – буква, нажатая перед щелчком.

60. Двойной щелчок левой кнопкой в неклиентской области окна выводит на экран другое окно. Щелчок правой кнопки в его клиентской области разрешает его перемещение вверх-вниз по нажатии F4.

61. Двойной щелчок правой кнопкой в неклиентской области окна разрешает перемещение окна по экрану с помощью клавиш: T – вверх, B – вниз, L – влево, R – вправо.

62. Щелчок левой кнопкой в неклиентской области окна приводит к созданию двух окон с разными стилями класса. Нажатие клавиш-стрелок позволяет перемещать любое окно, если при активном 1 окне нажата любая гласная буква.

63. Вывести на экран 2 окна. Щелчок левой кнопкой в неклиентской области любого окна изменяет заголовок второго окна. Нажатие любой цифры возвращает исходный заголовок.

64. Щелчок правой кнопкой в неклиентской области окна разрешает вывод двух дочерних окон. Первое окно выводится га экран при нажатии буквы «A», второе – при нажатии буквы «B».Закрытие всех окон – по нажатию «ESC».

65. Двойной щелчок левой кнопки в клиентской области окна выводит дочернее окно. Закрытие дочернего окна – нажатие F1, повторный вывод – нажатие F2.

66. Двойной щелчок правой кнопки в клиентской области окна разрешает перемещение окна влево, вправо, вверх и вниз соответственно при нажатии F1, F2, F8, F9.

67. Двойной щелчок левой кнопки в неклиентской области окна выводит на экран два дочерних окна. Нажатие любой цифры закрывает дочерние окна.

68. Двойной щелчок левой кнопки в неклиентской области окна приводит к появлению на экране дочернего окна. Заголовок этого окна может изменяться на букву – в зависимости от места щелчка.

69. Щелчок правой кнопки в клиентской области окна разрешает перемещать окно по экрану при нажатии клавиш-стрелок. Двойной щелчок левой кнопки запрещает такое перемещение.

70. Вывести на экран родительское окно и два дочерних. Щелчок левой кнопкой в клиентской области любого дочернего окна разрешает их перемещение по экрану вверх и вниз при нажатии букв H и D соответственно.

71. Двойной щелчок правой кнопкой в неклиентской области окна разрешает создание дочернего окна при нажатии цифр 0 или 1. Нажатие любой буквы уменьшает размеры дочернего окна в 2 раза.

72. В левом верхнем углу клиентской области окна создать дочернее окно. После нажатия левой клавиши мыши 4 раза «мигает» временное окно, а после нажатия правой – 3 раза «мигает» дочернее окно.

73. Создать окно приложения размером в одну шестнадцатую площади экрана с заголовком «Идет форматирование диска» без кнопок изменения размеров, закрытия и сворачивания в пиктограмму и без кнопки системного меню. При перемещении курсора мыши над клиентской областью окно должно «убегать» от курсора мыши в случайном образом выбранном направлении, оставаясь в пределах экрана.

74. В левом верхнем углу клиентской области окна создать временное окно площадью в одну шестнадцатую площади это области. При нажатии на левую (правую) клавишу мыши временное окно переместить в соседний по ходу (против хода) часовой стрелки угол клиентской области.

75. При запуске 2-го экземпляра приложения спросить пользователя, нужно ли его запустить. Если пользователь ответит «Да», то запустить его. Иначе изменить заголовок 1-го экземпляра.

76. Окно приложения без заголовка занимает весь экран фоном рабочего стола. Закрытие окна по щелчку правой кнопки мыши.

77. В центре клиентской области окна располагается невидимое окно без заголовка размером в четверть площади клиентской области. После нажатия левой клавиши мыши над клиентской областью любого из окон окно без заголовка должно стать видимым, а после нажатия правой – невидимым.

78. Создать окно размером в четверть площади экрана. После двойного щелчка мыши окно перемещается так, что его центр совпадает с координатами курсора мыши в момент щелчка.

79. Дочернее окно размером в ¼ родительского окна при перемещении курсора мыши над ним «убегает» от курсора мыши в произвольном направлении, оставаясь в пределах клиентской области родительского окна.

80. При запуске не первого экземпляра приложения выдать предупреждающее сообщение о количестве уже работающих копий этого приложения. Запустить экземпляр, только если согласен пользователь.

1.10 Контрольные вопросы

1. Какая информация содержится в одном сообщении?
2. Приведите пример предварительной обработки сообщения Windows.
3. Какие функции используются для получения дескрипторов стандартных ресурсов?

4. Какая информация указывается в структуре WNDCLASS?
5. Что такое стиль класса окна?
6. Поясните назначение стиля окна.
7. Зачем регистрировать класс окна?
8. Какие параметры имеет функция **CreateWindow**?
9. Поясните назначение функции **GetMessage**.
10. Какие параметры имеет функция **GetMessage?**
11. Зачем нужна функция **PeekMessage**?
12. При каком условии завершается цикл обработки сообщений?
13. Какие функции вызываются в теле цикла обработки сообщений минимального приложения?
14. В чем заключается отличие между функциями **PostMessage** и **SendMessage**?
15. Какие параметры имеет оконная функция?
16. Какие сообщения должна обрабатывать минимальная оконная функция?
17. Поясните структуру оконной функции.
18. Какое назначение имеет функция **DefWindowProc**?
19. Чем различаются аппаратные и символьные сообщения клавиатуры?
20. Какие системные и несистемные сообщения Вам известны?
21. Что передается в параметрах сообщений от мыши?
22. Что такое неклиентская область?
23. Какие макросредства MASM32 Вам известны?
24. Как выполнить захват и освобождение мыши?
25. Что такое время Windows?
26. Как организовать в программе выполнение действий с некоторой периодичностью?

2 ИНТЕРФЕЙС ГРАФИЧЕСКИХ УСТРОЙСТВ

2.1 Назначение и типы контекстов

Основой механизма взаимодействия приложений Windows с графическими устройствами (экран и принтер) является интерфейс графических устройств GDI (Graphics Device Interface). GDI – это совокупность программных средств Windows, организующих вывод на экран или принтер графических объектов – текстовых строк, геометрических фигур, изображений и т.п. Windows-приложение не имеет непосредственного доступа к аппаратуре. Вместо этого оно вызывает функции GDI, которые работают с драйверами физических устройств. Иначе говоря, приложения, обращаясь к функциям GDI, работают не с физическими устройствами вывода, а с логическими, и при вызове функций не учитывается физический способ отображения. Возможности вывода и аппаратные особенности устройства учитывает его драйвер. Благодаря этому механизму вывода графической информации приложения способны работать с любым устройством вывода, драйвер которого установлен в системе. Код библиотеки GDI находится в файле gdi32.dll. Драйверы стандартных устройств поставляются как часть подсистемы ввода-вывода Windows, а драйверы специализированных устройств предоставляются их производителями.

Примерами функциям GDI являются функции создания инструментов рисования (кисти, перья, шрифты), функции управления цветами, режимами рисования, функции вывода графических объектов. В задачу GDI входит контроль за границами выводимых объектов, чтобы они не затерли другие окна.

Параметры вывода на устройство должны быть установлены в **контексте устройства (**DC, Device Context) с помощью функций GDI. **Контекст устройства** – это системная структура данных, которая содержит характеристики устройства вывода и дескрипторы выбранных графических объектов и режимов рисования.

К основным графическим объектам относятся:

- Перо (pen) для рисования линий;.
- Кисть (brush) для заполнения фона или заливки фигур;
- Растровое изображение (bitmap);
- Палитра (palette) для определения набора доступных цветов;
- Шрифт (font) для вывода текста;
- Регион (region) для отсечения области вывода.

Рассмотрим понятие региона. **Регион** – это часть окна, с которой осуществляется работа. Понятно, что для ускорения вывода графики целесообразно в каждый момент времени изменять и обновлять только часть окна, а не перерисовывать все окно в целом. Существует несколько типов регионов.

Обновляемый *(update)*или ***недействительный****(invalid)* регион— это часть окна, которая требует обновления. *Видимый (visible)* регион — это та часть окна, которую в данный момент видит пользователь. Система изменяет видимый регион окна и в том случае, когда окно изменяет размеры, и в том случае, когда перемещение другого окна либо закрывает часть данного окна, либо открывает закрытую прежде часть. Регион ***отсечения*** *(clipping region)* ограничивает область, внутри которой разрешается отображение графической информации. Работа с регионами осуществляется с помощью функций **SetWindowRgn**, **SelectClipPath**, **SelectClipRgn**.

Если при вызове функции **CreateWindow** был использован стиль WS_CLIPCHILDREN или WS_CLIPSIBLINGS, то это вносит дополнительные правила в определение видимого региона, исключая из него любое дочернее или любые окна того же класса. Благодаря этому рисование не затрагивает отображаемые области таких окон.

Функции GDI используют только выбранные в контекст параметры и инструменты рисования. Например, для рисования линии заданной толщины в контексте в момент вызова функции рисования должен храниться дескриптор пера требуемой толщины. Функции вывода текста определяют размер, цвет, жирность шрифта по дескриптору шрифта. Если программисту не нужны свои значения в контексте, то для вывода будут использованы значения по умолчанию.

Контекст устройства также имеет свой дескриптор. Дескриптор контекста служит первым аргументом вызова всех функций, связанных с выводом в окно. Контекст относится к числу ресурсов Windows, которые сначала нужно запросить у системы, а после использования освободить.

Различают следующие типы контекстов устройства:

- контекст монитора;
- контекст принтера;
- контекст в памяти (совместимый контекст);
- метафайловый контекст;
- информационный контекст.

Контекст монитора создавать не нужно, его требуется только получить у ОС. Отметим, что контекст монитора может быть, в свою очередь, общим или частным. Общий тип контекста применяется, если

изменение содержимого контекста не очень интенсивно. Для графических редакторов, программ с интенсивным выводом графики используются окна с частным типом контекста дисплея, изменения в котором не пропадают даже после освобождения контекста. Для того чтобы окно имело частный контекст, нужно указать стиль класса CS_OWNDC.

2.2 Сообщение WM_PAINT и его обработка

Если окно перемещается по экрану с помощью клавиатуры или мыши, то сохранение в неизменном виде содержимого клиентской области окна обеспечивают системные программы. Если часть окна заслоняется при разворачивании пунктов меню, то временно закрытую область ОС также сохраняет сама и потом восстанавливает. Если же необходимо развернуть свернутое окно, или растянуть окно, или на фоне главного окна перемещается дочернее, то Windows уже не перерисовывает все окно, а посылает самому приложению сообщение WM_PAINT. Приложение, получив это сообщение, должно перерисовать содержимое клиентской области окна, содержимое же заголовка окна перерисовывает Windows. Обработка WM_PAINT включается в оконную функцию любого приложения, которое что-либо выводит на экран, Приложение может само информировать Windows, что ему требуется послать сообщение WM_PAINT. Это осуществляется с использованием функций **InvalidateRect**, **InvalidateRgn**, **UpdateWindow**. **InvalidateRect** объявляет заданную прямоугольную область окна поврежденной, что приводит к генерации Windows сообщения WM_PAINT. **UpdateWindow** передает WM_PAINT непосредственно в оконную функцию.

Рассмотрим классический алгоритм обработки в оконной функции сообщения WM_PAINT:

1) Получить у системы контекст устройства для окна. Для этого вызывается функция **BeginPaint**, имеющая два параметра. Первый – это дескриптор окна, в который будет направлен вывод изображения. Второй параметр – это адрес структуры PAINTSTRUCT, которую **BeginPaint** заполняет данными. Описание PAINTSTRUCT для MASM32 находится в **windows.inc**, а для C – в **winuser.h**.

```
PAINTSTRUCT STRUCT
  hDC    DWORD ?        ; дескриптор выделяемого контекста устройства
  fErase DWORD ?        ; флаг перерисовки фона окна
  rcPaint RECT <>       ; область вырезки
  fRestore DWORD ?      ; зарезервировано
  fIncUpdate DWORD ?    ; зарезервировано
```

rgbReserved BYTE 32 dup(?) ; зерезервировано
PAINTSTRUCT ENDS

Рассмотрим поля структуры. Если при заполнении структуры WNDCLASS не задать кисть для закрашивания фона окна, то функция **BeginPaint** установит флаг перерисовки окна ненулевым. Это означает, что приложение должно само закрашивать фон окна, иначе оно будет прозрачным. Этот режим практически не используется, и при задании кисти флаг перерисовки фона устанавливается равным нулю. Область вырезки представляет собой структуру **RECT**, описывающую прямоугольную область, которую необходимо перерисовать:

RECT STRUCT
left **dd ? ; X**-координата левого верхнего угла
top **dd ? ; Y**-координата левого верхнего угла
right **dd ? ; X**-координата правого нижнего угла
bottom **dd ? ; Y**-координата правого нижнего угла
RECT ENDS

Координаты области вырезки указываются относительно начала клиентской области окна, и, если перерисовывается вся клиентская область, то при расчете размеров вырезки из размеров окна вычитается толщина рамки и ширина заголовка. Перерисовка всей клиентской области – самый простой вариант, однако по координатам области вырезки можно реализовать перерисовку только испорченной области, что и делают функции GDI, даже если пользователь решил перерисовать все.

Функция **BeginPaint** возвращает дескриптор контекста монитора для клиентской области окна.

2) Установить свои режимы рисования или характеристики инструментов и сформировать требуемое изображение с помощью функций GDI, требующих контекст в качестве одного из параметров;

3) С помощью вызова функции **EndPaint** вернуть Windows контекст устройства, возвратив его в исходное состояние. Параметры **EndPaint** те же, что и у **BeginPaint**.

В программе на MASM32 это реализуется следующим образом:

```
        include \masm32\include\gdi32.inc
        includelib \masm32\lib\gdi32.lib
.....
B_WndProc:
        LOCAL hDC  :DWORD
        LOCAL Ps   :PAINTSTRUCT
```

```
.....
    .if Msg == WM_PAINT
        invoke BeginPaint,hWin,ADDR Ps
          mov hDC, eax
        invoke Paint_Proc,hWin,hDC
        invoke EndPaint,hWin,ADDR Ps
```

Если контекст монитора необходим вне обработки WM_PAINT, то его можно получить функцией **GetDC** (для клиентской области окна) или **GetWindowDC** (для всего окна), а освободить функцией **ReleaseDC.** Контекст, полученный с помощью **GetDC**, позволяет работать не с регионом, а со всей клиентской областью, и не требует объявления региона испорченным для генерации WM_PAINT, в отличие от **BeginPaint**.

Для получения дескриптора контекста **принтера** используется функция **CreateDC**, а для его освобождения – функция **DeleteDC**.

Научившись получать в программе дескриптор контекста устройства, рассмотрим, как вывести на монитор графическую информацию.

2.3 Описание инструментов рисования

Как было отмечено выше, к графическим объектам, обеспечивающим выполнение графических операций, относятся перья, кисти, растровые изображения, палитры, шрифты. Рассмотрим общий алгоритм работы с инструментами рисования:

а) создать новый инструмент с заданными характеристиками с помощью функций **Create…** (например, **CreatePen** (перо)) и запомнить его дескриптор;

б) использовать функцию **SelectObject** для загрузки в контекст устройства дескриптора созданного инструмента с предварительным сохранением дескриптора старого инструмента, который был установлен по умолчанию, если это первая смена инструмента. Удобство функции **SelectObject** в том, что она возвращает дескриптор того инструмента из контекста, который будет заменен, что используется для его сохранения;

в) использовать новый инструмент для вывода изображения или текста;

г) выбрать в контекст устройства сохраненный дескриптор инструмента по умолчанию;

д) уничтожить созданные инструменты функцией **DeleteObject**.

Создание новых инструментов обычно осуществляется при обработке сообщения WM_CREATE создания окна, для которого необхо-

димы новые инструменты.

Выбор инструментов и рисование осуществляются при обработке WM_PAINT, когда уже известен контекст устройства после отработки **BeginPaint**. Так как в контексте устройства хранится только один дескриптор инструмента рисования, то при смене кисти или пера в процессе рисования необходимо загружать в контекст дескрипторы требуемых инструментов по мере надобности. Сохранять при этом нужно только дескриптор инструмента по умолчанию, чтобы его можно было восстановить в контексте перед вызовом функции **EndPaint**.

Удаление созданных инструментов, как правило, реализуется при обработке сообщения WM_DESTROY, хотя можно создавать и удалять инструменты при обработке сообщения WM_PAINT.

Отметим, что в Windows предусмотрен стандартный набор кистей, перьев и шрифтов, которые могут быть использованы в программе. Их дескрипторы создавать и уничтожать не нужно, их нужно затребовать для использования функцией **GetStockObject** и затем загрузить в контекст.

Любая функция рисования линий и кривых, а также контуров замкнутых фигур использует **перо** (pen), дескриптор которого хранится в контексте устройства в данный момент. По умолчанию используется перо BLACK_PEN. Оно рисует сплошные черные линии толщиной 1 пиксел независимо от режима отображения. Кроме черного пера существует стандартное белое перо с шириной в 1 пиксел (WHITE_PEN), невидимое перо для вывода фигур без контура (NULL_PEN). Для рисования линий разного цвета, толщины и стиля GDI позволяет создавать *логические* ***перья***. Логическое перо имеет дескриптор типа HPEN *(handle to a pen).* Будем рассматривать так называемые простые перья. Для них не сплошные линии можно рисовать только толщиной 1 пиксел, а сплошные линии могут иметь любую толщину, но закругленные окончания.

Для получения дескриптора простого пера вызываются функция **CreatePen** или **CreatePenIndirect.** Функция **CreatePen** имеет три параметра:

1. ***Стиль пера***, определяющий порядок следования пикселов и расположение линии. Возможные значения этого параметра приведены в таблице 2.1.
2. ***Толщина линии*** в логических единицах. Если в контексте устройства используется режим отображения по умолчанию MM_TEXT, то логические единицы совпадают с физическими единицами и выражаются в пикселах. Если логические единицы не совпада-

ют с физическими, то физическая толщина пера устанавливается в соответствии с масштабом, и толщина пера будет одинаковой только в том случае, когда масштабы по обеим осям одинаковы. Если второй параметр равен нулю, то толщина принимается равной 1 пикселу независимо от режима отображения.

Таблица 2.1

Стиль	Вид линии	Выравнивание	Ограничения на толщину
PS_SOLID	Сплошная	По центру	Нет
PS_DASH	Пунктирная	По центру	<=1
PS_DOT	Точечная	По центру	<=1
PS_DASHDOT	Пунктирно-точечная	По центру	<=1
PS_DASHDOTDOT	Отрезок и две точки	По центру	<=1
PS_NULL	Не рисуется		
PS_INSIDEFRAME	Сплошная	Внутри контура	>1

3. ***Цвет пера***, имеющий тип COLORREF. Обычно это значение задается либо с помощью макроса RGB, либо с помощью макроса PALETTERGB. Первый вариант используется, если устройство вывода поддерживает полный диапазон цветов, определяемый 24-битным RGB-значением. Аргументами макроса являются интенсивности красной, зеленой и синей компонент цвета. Второй вариант (макрос PALETTERGB) необходимо использовать, если приложение работает с логической палитрой, например, для моделей дисплеев, которые поддерживают только 256 цветов. В последнем случае система Windows преобразует запрошенный RGB-цвет в наиболее близкий цвет из палитры.

Получение дескриптора пера с помощью **CreatePenIndirect** требует заполнения структуры типа LOGPEN, включающей поля, описывающие стиль, толщину и цвет пера.

После создания дескриптора пера с требуемыми параметрами его необходимо загрузить в контекст устройства с помощью функции SelectObject. Для рисования простейших геометрических фигур можно использовать следующие функции:

- **MoveToEx** – перемещение текущей позиции пера в точку (x,y). Параметры - дескриптор контекста устройства, x-координата новой текущей позиции, y-координата новой текущей позиции, предыдущая позиция пера. По умолчанию текущая позиция пера установлена в точку (0,0);

- **GetCurrentPositionEx** – получение текущей позиции пера. Параметры – hDC - дескриптор контекста устройства, адрес переменной pt типа POINT;

- **LineTo** - рисование отрезка, начиная с точки, в которой находится текущая позиция пера, до точки (xEnd, yEnd), не включая ее в отрезок. Параметры - дескриптор контекста устройства, xEnd - x-координата конечной точки, yEnd - y-координата конечной точки;

- **Polyline** – рисование ломаной линии. Параметры – hDC, адрес массива точек, количество точек. Функция **Polyline** не использует текущую позицию пера и не изменяет ее;

- **Дуги** рисуются как часть эллипса. Размеры и расположение эллипса определяются ограничивающим прямоугольником. Ограничивающий прямоугольник задается координатами левой верхней (xLeft, yTop) и правой нижней (xRight, yBottom) вершин. Центром эллипса является точка (x0, y0), где x0=xLeft + (xRight - xLeft)/2, а y0= - yTop + (yBottom - yTop)/2. Для рисования дуг предназначены функции **Arc**, **ArcTo** и **AngleArc.** В качестве параметров помимо координат точек, задающих эллипс, им передаются координаты начальной и конечной точек дуги;

Отметим, что существует функция вывода одного пиксела **SetPixel**, однако подобный способ рисования был бы очень медленным.

Все функции рисования, рассмотренные выше, рисуют незамкнутые фигуры. Однако хотелось бы иметь возможность рисовать замкнутые фигуры и закрашивать их внутренние области. Для этой цели используется кисть. Кисть представляет собой картинку размером 8*8 пикселов, которая при закрашивании дублируется по горизонтали и вертикали. К стандартным кистям относятся BLACK_BRUSH (черная), DKGRAY_BRUSH (темно-серая), GRAY_BRUSH (серая), LTGRAY_BRUSH (светло-серая) WHITE_BRUSH (белая, используется по умолчанию), NULL_BRUSH (пустая, без заливки), DC_BRUSH (сплошная цветная, по умолчанию белая, другой цвет после вызова **SetDCBrushColor**). Дескрипторы всех стандартных кистей запрашиваются по **GetStockObject**.

Для получения дескриптора нестандартной кисти используются следующие функции:

1. **CreateSolidBrush** - однотонная кисть. Параметр – цвет, фор-

мируемый RGB или PALETTERGB;

2. **CreateHatchBrush** - штриховая кисть. Параметры – стиль штриховки (HS_HORIZONTAL, HS_VERTICAL, HS_BDIAGONAL, HS_CROSS и т.д.), цвет.
3. **CreatePatternBrush** - кисть с произвольным рисунком в виде растрового изображения. Параметр – дескриптор растрового изображения.

Для создания кисти любого типа можно использовать функцию **CreateBrushIndirect**.

Рисование замкнутых фигур осуществляется с помощью функций:

- **Rectangle, FillRect, FrameRect, InvertRect** и **DrawFocusRect** - прямоугольник;
- **Ellipse** – эллипс;
- **Chord** – хорда;
- **Pie** – сектор эллипса;
- **RoundRect** – прямоугольник с закругленными углами;
- **Poligon** – многоугольник.

2.4 Работа со шрифтами и вывод текстовой информации

Любой шрифт характеризуется следующими параметрами**:**

- ***гарнитура*** *(typeface)* - совокупность нескольких начертаний символов шрифта, имеющих одинаковый стиль. Примерами гарнитур являются Arial, Times New Roman, Courier New**;**
- ***размер шрифта*** - высота прямоугольника, в котором помещается любой символ шрифта**;**
- ***начертание.*** Известны нормальное (normal) начертание, курсивное (italic), полужирное (bold), с подчеркиванием символа (underline), с перечеркиванием символа (strikeout). Виды начертаний могут комбинироваться в любом сочетании;
- *фиксированная* или *переменная* ***ширина*** символов. Шрифты первой группы называют *моноширинными.* В них все символы имеют одинаковую ширину. Шрифты второй группы называют *пропорциональными.* Пропорциональные шрифты чаще используются в текстовых документах**.**
- ***глиф*** *(glyph)* — графическая форма отдельного символа при его изображении. Различные гарнитуры различаются прежде всего глифами символов.

2.4.1 Типы шрифтов

Windows поддерживает две главные категории шрифтов: шрифты GDI и шрифты устройства *(device fonts).* Шрифты GDI хранятся в файлах, которые обычно расположены в одном из подкаталогов операционной системы. Шрифты устройства соответствуют конкретному устройству вывода. Например, большинство принтеров имеет набор встроенных шрифтов устройства. Шрифты GDI подразделяются на три типа:

- растровые шрифты;
- векторные шрифты;
- шрифты типа TrueType.

В шрифтах *растрового типа* символы хранятся в виде растровых картинок — прямоугольных матриц из точек-пикселов. Для каждого размера шрифта и, возможно, для различных разрешений экрана необходимо хранить свой набор символов. Растровые шрифты плохо масштабируются, так как при увеличении символов просто дублируются строки или колонки пикселов, что приводит к искажению очертаний глифов. Растровые шрифты применяются для воспроизведения текстовых элементов интерфейса Windows.

В *векторных шрифтах* глиф описывается последовательностью линейных отрезков, которые рисуются с помощью пера. Векторные шрифты легко масштабируются, однако имеют плохую четкость при маленьких размерах, а при большом увеличении контуры являются тонкими линиями. В настоящее время они применяются в основном только для плоттеров. Векторные шрифты хранятся в файлах.

TrueType — это технология контурных шрифтов, которая была разработана Apple Computer Inc. и Microsoft Corporation. Отдельные символы шрифтов TrueType определяются контурами, состоящими из прямых линий и кривых. При масштабировании контуры глифов остаются плавными. Когда в программе необходимо использовать шрифт TrueType определенного размера, Windows формирует растровое представление этого шрифта. Этот процесс называется *растеризацией.* Шрифт TrueType обычно хранится в одном файле с расширением .ttf.

2.4.2 Установка и удаление шрифтов в системе

Чтобы приложение смогло выводить текст, используя глифы некоего конкретного шрифта, он должен либо быть установлен в системной таблице шрифтов, либо быть встроенным шрифтом используемого

графического устройства. Имена шрифтов, установленных на графическом устройстве и хранящихся во внутренней системной таблице, можно получить при помощи функции **EnumFontFamilies** или **ChooseFont**.

Приложение может загрузить шрифт вызовом одной из функций: **AddFontResource** или **AddFontResourceEx**. Эти функции загружают шрифт из соответствующего ресурсного файла. Однако такая установка является временной, поскольку после рестарта операционной системы шрифт окажется недоступным. Чтобы установленный шрифт присутствовал в системе постоянно, информация о нем должна быть включена в реестр Windows.

Если установленный шрифт становится ненужным, то он может быть удален из системной таблицы с помощью функции **RemoveFontResource.**

Приложение, изменяющее системную таблицу шрифтов, должно оповестить об этом все окна верхнего уровня рассылкой сообщения WM_FONTCHANGE. Приложения, использующие список установленных шрифтов, должны обрабатывать это сообщение и обновлять содержимое списка шрифтов.

2.4.3 Метрики физического шрифта

При форматировании текста Win32 GDI использует *метрики шрифта,* которые поясняются на рисунке 2.1.

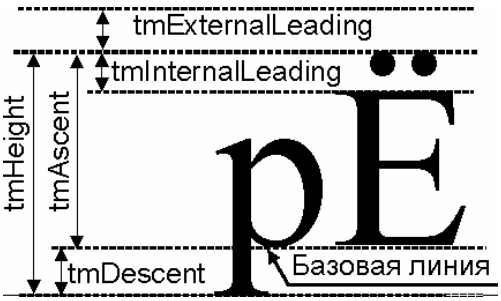


Рисунок 2.1

Отсчет всех размеров выполняется от *базовой линии* шрифта. На ней находится нижняя граница глифов большинства прописных букв. *Высота шрифта* tmHeight складывается из надстрочного интервала и подстрочного интервала. *Надстрочный интервал* tmAscent — это рас-

стояние от базовой линии до верхней границы ячейки символа. *Подстрочный интервал* tmDescent — это расстояние от базовой линии до нижней границы ячейки символа. *Внутренний зазор* tmInternalLeading определяет пространство для размещения диакритических знаков. *Внешний зазор* tmExternalLeading определяет минимальный интервал между соседними строками для многострочного текста.

В полиграфии размер шрифта измеряется в пунктах. Один пункт равен 0,01389 дюйма (1/71,99424). В компьютерной верстке пункт округляется до 1/72 дюйма. Еще один типографский термин «кегль» определяется как разность высоты шрифта и внутреннего зазора.

При выводе на экран текста учитываются характеристики (метрики) шрифта, описываемые структурой TEXTMETRIC, в которой указано 20 параметров. Получить данные из этой структуры можно функцией **GetTextMetrics**(). Возвращаемая функцией информация размещается в структуре типа TEXTMETRIC:

```
    typedef struct tagTEXTMETRIC {
    LONG tmHeight;       // высота символов
    LONG tmAscent;       // надстрочный интервал, т.е. часть высоты
// букв от базовой линии с учетом таких элементов, как две точки в
// букве "Ё"
    LONG tmDescent;      // подстрочный интервал, т.е. часть высоты
// букв ниже базовой линии
    LONG tmlnternalLeading;   // внутренний зазор, т.е. высота таких
// выступающих элементов, как две точки в букве "Ё". Может быть
// равен нулю
    LONG tmExternalLeading; // межстрочный интервал
    LONG tmAveCharWidth; // средняя ширина строчных букв,
// равна ширине латинской буквы "x"
    LONG tmMaxCharWidth; // максимальная ширина символов
    LONG tmWeight; // жирность (насыщенность) шрифта, может
// находиться в пределах от 0 до 1000, 400 для нормального
// начертания шрифта, или 700 для полужирного начертания
    LONG tmOverhang; // величина, на которую увеличивается
// ширина символов для наклонных или жирных шрифтов,
// получаемых из нормального шрифта. Шрифты TrueType обычно
// не используют это поле, так как в них для каждого  начертания
// создается свой шрифт
    LONG tmDigitizedAspectX; // разрешение устройства
// отображения по горизонтали
    LONG tmDigitizedAspectY; // разрешение устройства
```

// отображения по вертикали
TCHAR tmFirstChar; // код первого символа, для которого в
// шрифте имеется глиф
TCHAR tmLastChar; // код последнего символа, для которого в
// шрифте имеется глиф
TCHAR tmDefaultChar; // символ для замены символов, не
// имеющих глифа, обычно это контурный прямоугольник
TCHAR tmBreakChar; // символ для вставки в промежутки при
//выравнивании текста, обычно это пробел
BYTE tmltalic; // признак наклонности шрифта
BYTE tmUnderlined; // признак подчеркнутости шрифта
BYTE tmStruckOut; // признак перечеркнутости шрифта
BYTE tmPitchAndFamily; // Четыре младших бита указывают на шаг и технологию шрифта (моноширинный или пропорциональный – бит 0, векторный шрифт – бит 2, шрифт TrueType – бит 3, шрифт устройства – бит 4). Четыре старших бита - указывают семейство шрифта (например, Roman, Modern, Swiss).
BYTE tmCharSet; // код набора символов (ANSI_CHARSET, SYMBOL_CHARSET, RUSSIAN_CHARSET и т.д.)
}TEXTMETRIC;

2.4.4 Логические шрифты. Функции вывода текста и изменения цветовых характеристик

Приложения работают не с физическими, а с логическими шрифтами. Логический шрифт – это объект GDI, содержащий требования к шрифту со стороны приложения. Эти требования анализируются подсистемой GDI, и с помощью драйверов шрифтов подбираются подходящие зарегистрированные в системе физические шрифты. Логический шрифт имеет дескриптор. По умолчанию в контекст устройства загружен растровый шрифт с символами переменной ширины SYSTEM_FONT с кодировкой ANSI. Существует набор встроенных шрифтов, которые дескрипторы которых запрашиваются для загрузки в контекст функцией **GetStockObject.**

Если требуется создать собственный логический шрифт, то используется функция **CreateFont** или **CreateFontIndirect**. Функция **CreateFont** имеет 14 параметров, описывающих логический шрифт, а **CreateFontIndirect** – один параметр, представляющий собой указатель на структуру **LOGFONT**, содержащую те же 14 параметров логического шрифта**.**

Структура **LOGFONT** имеет поля:

- *lfHeight* – желательная высота шрифта (по умолчанию 12 пунктов);

- *lfWidth* – желательная средняя ширина символа;

- *lfEscapement* – угол между базовой линией текста и осью OX устройства (отсчет против часовой стрелки);

- *lfOrientation* – ориентация символа, т.е. угол между базовой линией символа и осью OX устройства (отсчет против часовой стрелки). Это поле используется независимо от lfEscapement только в расширенном графическом режиме, в остальных случаях значения ориентации символа совпадает с lfEscapement;

- *lfWeight* – желательная жирность логического шрифта (от 0 до 1000);

- *lfItalic, lfUnderline, lfStrikeOut* – признаки курсива, подчеркнутого, перечеркнутого шрифта;

- *lfCharSet* – набор символов логического шрифта;

- *lfOutPrecision* – критерий соответствия параметров логического шрифта имеющимся физическим шрифтам при их подборе. Например, возможно указать предпочтительный выбор растровых или TrueType шрифтов;

- *lfClipPrecision* – метод изменения изображения символа, попавшего за пределы региона отсечения;

- *lfQuality* – качество вывода глифов (0 – внешний вид неважен, 1 – позволяет изменять растровые шрифты с потерей качества, 2 – запрещает изменение растровых шрифтов, 3 и 4 – запрещает/ разрешает сглаживание);

- *lfPitchAndFamily* – шаг и семейство логического шрифта;

- *lfFaceName* – строка, содержащая имя гарнитуры шрифта, заканчивающаяся нулем. Длина строки не должна превышать 32 байта. Имена имеющихся гарнитур можно получить с помощью функции **EnumFontFamiliesEx**.

После создания логического шрифта функциями **CreateFont** или **CreateFontIndirect** ими возвращается дескриптор шрифта, который необходимо перед выводом текста загрузить в контекст устройства функцией **SelectObject**. При этом происходит подстановка физического шрифта. GDI сравнивает параметры логического шрифта с параметрами различных шрифтов, доступных для графического устройства, выбирая наиболее подходящий шрифт. Для сравнения используются штрафные очки, которые имеют разные весовые коэффициенты. Выбирается тот шрифт, для которого штрафная сумма будет наименьшей. Наиболее важным фактором при подборе физического шрифта является набор символов, который задается в поле IfCharSet. При несовпаде-

нии этого атрибута очень велика вероятность, что символы будут выводиться совершенно неверными глифами. Следующее по важности поле — это LfOutPrecision. Этот показатель ограничивает рассматриваемые наборы символов определенными типами шрифтов. Затем оценивается поле IfFaceName, а после него — поле IfPitchAndFamily. После сравнения указанных полей GDI сравнивает высоту символов, заданную в поле IfHeight, а затем поля IfWidth, Ifltalic, IfUnderLine, LfStrikeOut.

Для вывода текстовой строки в простейшем случае можно использовать **TextOut**, имеющую следующие параметры:

- дескриптор контекста устройства;
- x-координата начальной точки вывода;
- y-координата начальной точки вывода;
- адрес выводимой строки;
- число символов в строке.

Позиционирование строки зависит от текущего режима выравнивания, заданного в контексте устройства и определяющего, что считать опорной точкой при выводе – заданную начальную точку или текущую позицию пера в контексте устройства. Также режим выравнивания определяет, как позиционировать строку текста (обрамляющий прямоугольник) относительно опорной точки и как выводить текст: слева направо или справа налево. Значение этого атрибута можно изменить при помощи функции **SetTextAlign**.

При выводе текста можно изменить цвет символов (**SetTextColor**), цвет фона под символами (**SetBkColor**), режим фона знакомест (прозрачный или нет) (**SetBkMode**), выравнивание текста относительно заданных в TextOut координат. Для вывода текста также можно использовать функцию **DrawText**, позволяющую выводить длинный текст в заданный прямоугольник.

2.5 Вывод растровых изображений

2.5.1 Типы растров

Растр представляет собой набор пикселов, каждому из которых сопоставлены биты, кодирующие его цвет. Количество цветов пиксела равно 2^N, где N – количество битов, отводимых для хранения цвета. Например, для режимов TrueColor для хранения цвета используется 24 бита, по 8 бит на каждую из компонент RGB, а в режимах HiColor цвет пиксела хранится в 16 битах.

Если устройство вывода поддерживает полный диапазон цветов,

определяемых битами, соответствующими пикселу, то никакой дополнительной информации для вывода растра использовать не нужно. Однако если устройство вывода поддерживает только ограниченный набор цветов, то требуется согласовывать цвета при выводе, для чего используется палитра. Цветовая палитра – это массив цветов, которые способно отображать устройство. Различают системную и логическую палитры.

Системная палитра – это объект ОС, задающий цвета, которые могут одновременно отображаться устройством. Системная палитра зависит от аппаратных возможностей устройства отображения. Для монитора она содержит 256 элементов, 20 из которых зарезервировано для статических цветов, которые не изменяются и используются для вывода элементов стандартного интерфейса.

Логическая палитра должна быть сначала создана в приложении функцией **CreatePalette**, а затем должна быть адаптирована для использования функцией **RealizePalette**, которая занимает неиспользуемые элементы системной палитры элементами логической, или, если неиспользуемых элементов не хватает для всех цветов логической палитры, подбирает смешивание имеющихся цветов. Формирование цвета для палитры осуществляется с помощью макроса PALETTERGB, параметрами которого являются интенсивности красной, зеленой и синей компонент цвета. Каждое приложение может реализовать свою логическую палитру, поэтому перед предоставлением приложению фокуса ввода ему посылается сообщение, обработка которого восстанавливает цвета, измененные другими приложениями. По умолчанию логическая палитра содержит 20 базовых цветов.

В настоящее время различают аппаратно-зависимые (**DDB**, Device Dependent Bitmap) и аппаратно-независимые (**DIB**, Device Independent Bitmap) растры.

DDB-растры используются как внутренний формат хранения графики в графической подсистеме ОС и для хранения изображений, не предназначенных для вывода на другой аппаратной платформе. Для DDB-растра ОС всегда создает битовый образ в памяти с учетом параметров конкретного графического устройства. Формат DDB не подходит для переноса растров на другие компьютеры.

DIB-растры, помимо массива пикселов, содержат цветовую таблицу и справочную информацию. Цветовая таблица является массивом структур, каждая из которых содержит 3 байта для интенсивностей компонент RGB и 1 резервный байт. Для пиксела же указывается номер элемента в таблице цветов. DIB-растры используются для обмена изображениями между приложениями. Для хранения DIB-растров ис-

пользуется файловый формат BMP.

2.5.2 Технология отображения растров

Для вывода DIB-растров используются функции **StretchDIBits** и **SetDIBitsToDevice**. Функция **StretchDIBits** имеет следующие параметры:

- дескриптор контекста устройства;
- x и y координаты, ширина и высота области-приемника изображения (в логических единицах);
- x и y координаты, ширина и высота области-источника изображения(в пикселах);
- адрес массива пикселов изображения;
- адрес заголовка информационного блока в bmp;
- флаг интерпретации цветовой таблицы;
- код растровой операции.

Область-источник указывает, какая часть растра выводится, происходит ли зеркальное отображение по горизонтали и/или по вертикали. Область-приемник задает место на поверхности изображения графического устройства, область-приемник также можно зеркально отразить и масштабировать. Если размеры областей источника и приемника совпадают, то вывод растра происходит без масштабирования. Способ добавления новых или удаления ненужных пикселов хранится в контексте устройства и может быть изменен функцией **SetStretchBltMode**. Так, возможно вычислять средний цвет по группе пикселов, комбинировать пикселы с помощью побитовых логических операций, отбрасывать лишние пикселы.

Флаг интерпретации цветовой таблицы указывает, что именно содержит эта таблица – компоненты RGB или индексы цветов в логической палитре.

Код растровой операции показывает способ копирования из области-источника в область-приемник. Простейшим кодом является SRCCOPY, задающий копирование без дополнительных операций.

Функция **SetDIBitsToDevice** позволяет выводить растры с сохранением масштаба и ориентации либо полностью, либо по строкам.

Для работы с DDB-растрами, являющимися объектами GDI и хранящимися в памяти GDI, используются другие API-функции.

Создать DDB-растр можно с помощью функций **CreateBitmap**, **CreateBitmapInDirect**, **CreateCompatibleBitmap**. Все эти функции возвращают дескриптор растрового объекта. **CreateBitmap** имеет следующие параметры;

- ширина и высота растра в пикселах;
- количество плоскостей *cPlanes* для хранения цвета пиксела;
- количество битов на 1 пиксел *cBitsToPxl*;
- адрес массива с пикселами.

По параметрам *cPlanes* и *cBitsToPxl* драйвер графического устройства осуществляет подбор формата хранения цвета, чтобы количество битов на пиксел не превышало произведения указанных параметров.

Функция **CreateBitmapInDirect** работает с теми же параметрами, что и **CreateBitmap**, однако они передаются через структуру типа BITMAP. Функция **CreateCompatibleBitmap** создает DDB-растр, совместимый с контекстом заданного устройства. Недостатком трех рассмотренных функций является то, что они создают либо монохромный растр с инициализацией пикселов, либо цветной растр без инициализации.

Для работы в программе с инициализированными цветными растрами лучшим выходом является использование ресурсов. Ресурсами являются пиктограммы, рисунки, меню, диалоговые окна, курсоры и шрифты. Они описываются во внешнем текстовом файле, называемом файлом ресурсов и имеющим расширение **rc**. Различные ресурсы описываются в нем в соответствии со своими форматами. Для того чтобы включить ресурсы в *.exe файл, их необходимо откомпилировать специальным компилятором ресурсов, который может быть встроен в интегрированную среду разработки программ. Результатом работы компилятора ресурсов является файл с расширением **res**, который вместе с объектным модулем обрабатывается редактором связей для получения exe-файла. Преимуществами использования ресурсов являются простота их редактирования и независимость описания от текста исходного модуля, возможность редактирования ресурсов непосредственно в загрузочном файле (например, для перевода пунктов меню на другой язык), возможность извлечения ресурсов из загрузочного файла для последующего использования в других приложениях.

Каждому ресурсу при описании присваивается идентификатор ресурса, который затем используется API-функциями. Например, описание пиктограммы ICON1.ICO имеет вид:

```
Идентификатор_ресурса ICON [параметры] "ICON1.ICO"
```

а описание растрового изображения BITMAP1.BMP:

```
Идентификатор_ресурса BITMAP [параметры] "BITMAP1.BMP"
```

В MASM32 используется числовой идентификатор ресурса.

Несмотря на важность DDB-растров, в Windows отсутствует функция их отображения на устройстве вывода. Их вывод осуществляется

путем копирования с одного устройства на другое как без изменения масштаба (функция **BitBlt**), так и с изменением (функция **StretchBlt**). Устройством-приемником служит окно, а в качестве устройства-источника в памяти нужно создать область, которая по своим возможностям должна быть совместима с экраном и в которую нужно записать изображение. Совместимая с окном область памяти также должна иметь свой контекст, называемый контекстом памяти или совместимым контекстом памяти. В контексте памяти хранятся дескрипторы инструментов рисования, которые можно заменять на свои, в контекст памяти можно выводить графику. Отличием контекста памяти от контекста окна является возможность загрузки в контекст памяти дескриптора растрового изображения.

Рассмотрим реализацию вывода на экран DDB-растра, описанного как ресурс. Для вывода растрового изображения необходимо выполнить следующие действия:

1) загрузить растровое изображение в память и запомнить его дескриптор. Для этого используется функция **LoadBitmap**, первым параметром которой является дескриптор экземпляра приложения, а вторым – идентификатор ресурса, описывающего DIB-растр. Функция выделяет память под изображение, загружает его туда, преобразует в DDB-формат и возвращает дескриптор hBmp области памяти с растром. Этот шаг выполняется вне обработки сообщения WM_PAINT. Отметим, что если требуется работать с изображением, размер которого будет известен позже, то можно зарезервировать требуемый объем памяти функцией **CreateCompatibleBitmap**. Загрузку битового образа можно осуществлять с помощью функции **LoadImage**, загружающей не только растры, но еще пиктограммы и курсоры. **LoadImage** возвращает дескриптор DDB-растра.

2) получить совместимый с окном контекст устройства для области памяти, где будет храниться изображение, с использованием функции **CreateCompatibleDC**, единственным параметром которой является дескриптор контекста устройства, куда должно быть выведено изображение. Этот дескриптор hDC либо возвращается функцией **BeginPaint** в обработке WM_PAINT, либо должен быть получен с помощью функции **GetDC**. Если использовалась функция **GetDC**, необходимо после работы вернуть контекст системе с помощью функции **ReleaseDC**.

3) включить дескриптор области памяти с растровым изображением hBmp в совместимый с окном контекст памяти, используя функцию **SelectObject**. Значение дескриптора растрового изображения по умолчанию в совместимом контексте нужно сохранить, хотя он описывает

изображение размером всего в 1 пиксел. После этого в совместимую память можно выводить любые объекты, вызывать функции рисования.

4) Для того чтобы отобразить на экране созданное изображение, необходимо скопировать его в окно с использованием функций **BitBlt** или **StretchBlt**. Рассмотрим вызов и параметры функции **BitBlt.** Вызов имеет вид:

```
BitBlt(hDC,x,y,w,h,memDC,x0,y0,FLAG)
```

Параметры:

- **hDC** и **memDC**- дескрипторы окна приложения, куда копируется изображение, и совместимого контекста памяти;

- **x0** и **y0** – координаты начальной точки верхнего левого угла изображения-источника, с которой начинается копирование в изображение-приемник. Если все изображение копируется полностью, то нужно указать нулевые значения **x0** и **y0**;

- **x** и **y** задают координаты верхнего левого угла копии в окне, считая от начала его клиентской области;

- **w** и **h** задают ширину и высоту копируемой прямоугольной части изображения. Максимальное значение этих параметров – ширина и высота в пикселах исходного изображения. Характеристики изображения можно получить с помощью функции **GetObject**, которая для растрового изображения заполняет структуру **BITMAP**, поля которой **bmWidth** и **bmHeight** хранят ширину и высоту изображения.

- FLAG – указывает вид операции над битами изображения и битами области копирования, что используется для реализации мультипликации. В простейшем случае FLAG=SRCCOPY, что указывает на копирование источника в приемник с затиранием области копирования. Если нужно реализовать мультипликацию, то в цикле перемещения объекта сначала выводится изображение с FLAG=SRCCOPY, затем после задержки с FLAG=MERGEPAINT, что дает стирание изображения и получение белого фона, после чего изменяют координаты вывода.

Функция **StretchBlt** позволяет масштабировать изображение и выполнять его зеркальные отражения.

5) вернуть в совместимый контекст памяти старое значение дескриптора битового изображения.

6) Уничтожить контекст совместимой с окном области памяти.

Иллюстрирующий текст продедуры вывода DDB-растра на MASM32:

```
Paint_Proc proc hWin:DWORD, hDC:DWORD
   LOCAL hOld:DWORD
   LOCAL memDC :DWORD
   invoke CreateCompatibleDC,hDC
   mov memDC, eax
   invoke SelectObject,memDC,hBmp
   mov hOld, eax
   invoke BitBlt,hDC,10,10,166,68,memDC,0,0,SRCCOPY
   invoke SelectObject,memDC,hOld
   invoke DeleteDC,memDC
   return 0
Paint_Proc endp
```

2.6 Примеры вывода в клиентскую область окна графики и текста

Пусть в приложении необходимо выводить в клиентскую область окна текст, графические примитивы и растровое изображение. Для того чтобы можно было выполнять этот вывод независимо и в нужной последовательности, будем при нажатии клавиши «1» выводить строки текста, при нажатии клавиши «2» выводить графические фигуры, а при нажатии «3» - растровое изображение. Тогда выбор при выводе будем запоминать в переменной w при обработке сообщения об отпускании клавиши WM_KEYUP, а собственно вывод будем реализовывать в функциях **Font_Draw, Graf_Draw и Paint_Proc**:

```
    .elseif uMsg == WM_KEYUP
             .if wParam==VK_1
                   mov w,1
             .elseif wParam==VK_2
                   mov w,2
             .elseif wParam==VK_3
                   mov w,3
             .endif
             invoke InvalidateRect,hWin,NULL,TRUE
    .elseif uMsg == WM_PAINT
        invoke BeginPaint,hWin,ADDR Ps
          mov hDC, eax
          .if w==1
                   invoke Font_Draw,Ps.hdc
             .elseif w==2
                   invoke Graf_Draw,Ps.hdc
             .elseif w==3
```

```
                invoke Paint_Proc,hWin,hDC
            .endif
        invoke EndPaint,hWin,ADDR Ps
        return 0
```

Демонстрационная функция для вывода текста:

```
Font_Draw proc hDC :HDC
       LOCAL hObj :DWORD
       LOCAL hOldObj :DWORD
       szText szString,"*Assembler*"
       szText Font0,"Times New Roman"
       szText Font1,"Century Gothic"
       szText Font2,"Monotype Corsiva"
       RGB 34,57,94
       invoke SetBkColor,hDC,eax

;выведем текст OEM-шрифтом, дескриптор уже существует
       invoke GetStockObject,OEM_FIXED_FONT
       mov hObj,eax
       invoke SelectObject,hDC,hObj
       mov hOldObj,eax
       invoke TextOut,hDC,20,20,offset szString,11
       invoke SelectObject,hDC,hOldObj
       invoke DeleteObject,hObj

;выведем текст системным шрифтом
    invoke GetStockObject,SYSTEM_FONT
    mov hObj,eax
    invoke SelectObject,hDC,hObj
    mov hOldObj,eax
    invoke TextOut,hDC,450,450,offset szString,11
    invoke SelectObject,hDC,hOldObj
    invoke DeleteObject,hObj

;заполняем структуру типа LOGFONT
    mov Font.lfHeight,40
       mov Font.lfWidth,15
       mov Font.lfEscapement,150 ;наклон
       mov Font.lfOrientation,50 ;наклон букв
       mov Font.lfWeight,1000    ;толщина
       mov Font.lfItalic,FALSE
       mov Font.lfUnderline,TRUE
       mov Font.lfStrikeOut,FALSE
       mov Font.lfCharSet,ANSI_CHARSET ;начертание
       mov Font.lfOutPrecision,OUT_OUTLINE_PRECIS
       mov Font.lfClipPrecision,CLIP_DEFAULT_PRECIS
```

```
        mov Font.lfQuality,DRAFT_QUALITY
        mov Font.lfPitchAndFamily,0
        invoke lstrcpy, addr Font.lfFaceName,
                        addr Font0
; копируем имя гарнитуры шрифта из Font0
        invoke CreateFontIndirect,addr Font
        mov hObj,eax
        invoke SelectObject,hDC,hObj
        invoke SetTextColor,hDC,155
        invoke TextOut,hDC,250,100,offset szString,11
        invoke DeleteObject,hObj
; изменим параметры шрифта и цвет текста при выводе
        INVOKE lstrcpy, addr Font.lfFaceName,
                addr Font1
        mov Font.lfHeight,20
        mov Font.lfWidth,30
        mov Font.lfEscapement,0
        mov Font.lfUnderline,FALSE
        mov Font.lfStrikeOut,TRUE
        invoke CreateFontIndirect,addr Font
        mov hObj,eax
        invoke SelectObject,hDC,hObj
        invoke SetTextColor,hDC,249
        invoke TextOut,hDC,50,350,offset szString,11
        invoke DeleteObject,hObj
; еще раз изменим параметры шрифта и цвет текста
        invoke lstrcpy, addr Font.lfFaceName,
                addr Font2
        mov Font.lfHeight,20
        mov Font.lfWidth,15
        mov Font.lfUnderline,FALSE
        mov Font.lfWeight,100
        mov Font.lfStrikeOut,FALSE
        mov Font.lfItalic,TRUE
       invoke CreateFontIndirect,addr Font
        mov hObj,eax
        invoke SelectObject,hDC,hObj
        invoke SetTextColor,hDC,50
        invoke TextOut,hDC,60,160,offset szString,11
        invoke DeleteObject,hObj
        ret
Font_Draw endp
```

Демонстрационная функция для вывода графических примитивов:

```
Graf_Draw proc hDC:HDC
        LOCAL hObj:DWORD
```

```
LOCAL lb:LOGBRUSH
LOCAL p1:POINT
LOCAL p2:POINT
LOCAL p3:POINT
LOCAL p4:POINT

invoke MoveToEx,hDC,50,10,NULL
invoke LineTo,hDC,300,10

invoke CreatePen,PS_SOLID or PS_DOT,1,25
invoke SelectObject,hDC,eax
mov hObj,eax
invoke MoveToEx,hDC,50,30,NULL
invoke LineTo  ,hDC,300,30
invoke SelectObject,hDC,hObj
invoke DeleteObject,eax

mov lb.lbStyle,BS_HATCHED
mov lb.lbColor,9452311
mov lb.lbHatch,HS_BDIAGONAL
invoke CreateBrushIndirect,addr lb
invoke SelectObject,hDC,eax
invoke Rectangle,hDC,300,300,500,500
invoke Arc,hDC,200,200,300,90,310,90,310,590
invoke DeleteObject,eax

mov p1.x,250
mov p1.y,300
mov p2.x,550
mov p2.y,300
mov p3.x,400
mov p3.y,200
mov lb.lbStyle,BS_HATCHED
mov lb.lbColor,14950000
mov lb.lbHatch,HS_DIAGCROSS
invoke CreateBrushIndirect,addr lb
invoke SelectObject,hDC,eax
invoke Polygon,hDC,addr p3,3
invoke DeleteObject,eax

mov lb.lbStyle,BS_SOLID
mov lb.lbColor,00FF0000h
mov lb.lbHatch,0
invoke CreateBrushIndirect,addr lb
invoke SelectObject,hDC,eax
invoke Ellipse,hDC,375,230,425,280
invoke DeleteObject,eax
```

```
        invoke CreatePen,PS_SOLID,8,255000
        invoke SelectObject,hDC,eax
        mov hObj,eax
        mov lb.lbStyle,BS_HATCHED
        mov lb.lbColor,000000FFh
        mov lb.lbHatch,0
        invoke CreateBrushIndirect,addr lb
        invoke SelectObject,hDC,eax
        invoke Pie,hDC,10,120,110,220,350,120,90,120
        invoke Pie,hDC,10,120,110,220,90,430,90,200
        invoke Pie,hDC,10,120,110,220,-800,0,-800,800
        invoke DeleteObject,eax
        invoke SelectObject,hDC,hObj
        invoke DeleteObject,eax
        ret
Graf_Draw endp
```

Вывод растрового изображения для реализации эффекта мультипликации может осуществляться либо путем вывода меняющихся кадров в одно место клиентской области окна, либо организуется цикл, в котором изменяются координаты вывода растра. Рассмотрим первый из этих способов.

Пусть имеются 4 кадра img1.bmp, img2.bmp img3.bmp, img4.bmp из мультфильма, приведенные на рисунке 2.2. Эти растровые изображения опишем в файле ресурсов следующим образом:

```
100 BITMAP MOVEABLE PURE LOADONCALL DISCARDABLE "img1.bmp"
200 BITMAP MOVEABLE PURE LOADONCALL DISCARDABLE "img2.bmp"
300 BITMAP MOVEABLE PURE LOADONCALL DISCARDABLE "img3.bmp"
400 BITMAP MOVEABLE PURE LOADONCALL DISCARDABLE "img4.bmp"
```

Идентификаторы растровых изображений при таком описании – это числа 100, 200, 300, 400.

Для того чтобы реализовать вывод мультфильма в клиентскую область окна, необходимо организовать циклический перебор номеров картинок и вывод текущей картинки. Обозначим номер картинки через nPicture и опишем эту переменную в WinMain:

```
nPicture dw 0
```

img1.bmp

img2.bmp

img3.bmp

img4.bmp

Рисунок 2.2

Собственно перебор номеров картинок осуществляется при обработке сообщения WM_PAINT в оконной функции:

```
WndProc proc  hWin   :DWORD,
              uMsg   :DWORD,
              wParam :DWORD,
              lParam :DWORD
       LOCAL hDC    :DWORD
       LOCAL Ps     :PAINTSTRUCT
       LOCAL hOld   :DWORD
...
       .if uMsg == WM_CREATE
...
       .elseif uMsg == WM_PAINT
               invoke Sleep,200
; задержка, чтобы изображения не менялись очень быстро
; выбор номера следующей картинки так, чтобы номера
; изменялись циклически от 100 до 400
             .if nPicture == 400
                   mov nPicture,100
             .elseif
                   add nPicture,100
```

```
            .endif
            invoke LoadBitmap,hInstance,nPicture
; загрузим текущую картинку
            mov hBmp, eax
; запомним ее дескриптор
            invoke BeginPaint,hWin,ADDR Ps
; получим контекст устройства и запомним его в eax
            mov hDC, eax
; вызовем функцию рисования
            invoke Paint_Proc,hWin,hDC
; вернем контекст устройства системе
            invoke EndPaint,hWin,ADDR Ps
; обновление всей клиентской области окна hWnd
; недействительной
            invoke RedrawWindow,hWnd,NULL, NULL,
                   RDW_INVALIDATE
            return 0
      .elseif uMsg == WM_DESTROY
            invoke PostQuitMessage,NULL
; посылка сообщения WM_QUIT в очередь потока
      .endif
      invoke DefWindowProc,hWin,uMsg,wParam,lParam
      ret
WndProc endp
```

В функции PaintProc осущетвляется вывод текущего растра:

```
Paint_Proc proc hWin   :DWORD,
                hDC    :DWORD
      LOCAL   hOld   :DWORD
      LOCAL   memDC  :DWORD
; создаем совместимый контекст
      invoke CreateCompatibleDC,hDC
      mov memDC, eax
; выбираем в совместимый контекст дескриптор растра
      invoke SelectObject,memDC,hBmp
; и сохраняем дескриптор старого изображения
      mov hOld, eax
; выводим растр
      invoke BitBlt,hDC,238,30,352,288,memDC,0,0,SRCCOPY
; возвращаем в совместимый контекст дескриптор старого
; изображения
      invoke SelectObject,hDC,hOld
      invoke DeleteDC,memDC
      return 0
Paint_Proc endp
```

Для реализации вывода последовательности изображений в разных местах клиентской области можно использовать таймер и при обработке сообщения WM_TIMER в оконной функции изменять номер изображения и координаты вывода:

```
    .elseif uMsg == WM_TIMER
            inc nom_img
            .if nom_img == 500
                  mov nom_img, 100
            .endif
; увеличим координату x изображения
            inc x_img
; увеличим координату y изображения
            inc y_img
```

В функции вывода растра загружается изображение с номером nom_img и затем выводится:

```
      invoke LoadBitmap, hInstance ,nom_img
      mov img_BMP,eax
      invoke CreateCompatibleDC,hDC
      mov memDC, eax
      invoke SelectObject,memDC,img_BMP
      mov hOld, eax
      ;вывод изображения
      invoke
BitBlt,hDC,x_img,y_img,100,60,memDC,0,0,SRCCOPY
      invoke SelectObject,hDC,hOld
```

Для организации задержек при выводе растров можно использовать следующее макроопределение:

```
  delay MACRO num
      invoke GetTickCount
      mov dval, eax
      add dval, num
      .while eax < dval
            invoke GetTickCount
      .endw
  ENDM
```

Рассмотрим вывод движущегося объекта, например, автомобиля, причем при движении будем использовать несколько растров с автомобилем, которые отличаются углом поворота колес и наличием выхлопных газов.

Ресурсы описаны в файле rsrc.rc:

```
1 BITMAP MOVEABLE PURE LOADONCALL DISCARDABLE "car1.bmp"
2 BITMAP MOVEABLE PURE LOADONCALL DISCARDABLE "car2.bmp"
3 BITMAP MOVEABLE PURE LOADONCALL DISCARDABLE "car3.bmp"
4 BITMAP MOVEABLE PURE LOADONCALL DISCARDABLE "car4.bmp"
5 BITMAP MOVEABLE PURE LOADONCALL DISCARDABLE "car5.bmp"
6 BITMAP MOVEABLE PURE LOADONCALL DISCARDABLE "car6.bmp"
```

Библиотеки, прототипы функций и макроопределения описаны в файле l.inc:

```
; включаемые файлы
   include c:\MASM32\INCLUDE\windows.inc
   include c:\MASM32\INCLUDE\masm32.inc
   include c:\MASM32\INCLUDE\gdi32.inc
   include c:\MASM32\INCLUDE\user32.inc
   include c:\MASM32\INCLUDE\kernel32.inc
; включаемые библиотеки
   includelib c:\MASM32\LIB\masm32.lib
   includelib c:\MASM32\LIB\gdi32.lib
   includelib c:\MASM32\LIB\user32.lib
   includelib c:\MASM32\LIB\kernel32.lib
; прототипы функций
   WinMain PROTO :DWORD
   WndProc PROTO :DWORD,:DWORD,:DWORD,:DWORD
   Paint_Proc   PROTO :DWORD,:DWORD , :DWORD
   Pause PROTO: DWORD
; Макросы
   szText MACRO Name, Text:VARARG
     LOCAL lbl
      jmp lbl
       Name db Text,0
      lbl:
   ENDM
   m2m MACRO M1, M2
      push M2
      pop  M1
   ENDM
   return MACRO arg
      mov eax, arg
      ret
   ENDM
.data
   szDisplayName db "Пример ",0
   sze           dd 0
```

```
      var3          dd 0
      hWnd          dd 0
      hInstance     dd 0
      hBmp          dd 0
      x_x           dd 0
   ; координата автомобиля
```

Текст программы, реализующей вывод растровых изображений:

```
.386
.model flat, stdcall
option casemap :none
include l.inc
.code
start:
      invoke GetModuleHandle, NULL
      mov hInstance, eax
      invoke WinMain,hInstance
      invoke ExitProcess,eax
; главная функция
WinMain proc hInst:DWORD
      LOCAL wc   :WNDCLASSEX
      LOCAL msg  :MSG
      szText szClassName,"myClass"
      mov wc.cbSize,         sizeof WNDCLASSEX
      mov wc.style,          CS_BYTEALIGNWINDOW
      mov wc.lpfnWndProc,    offset WndProc
      mov wc.cbClsExtra,     NULL
      mov wc.cbWndExtra,     NULL
      m2m wc.hInstance,      hInst
      mov wc.hbrBackground,  COLOR_WINDOW
      mov wc.lpszMenuName,   NULL
      mov wc.lpszClassName,  offset szClassName
      m2m wc.hIcon,          NULL
      invoke LoadCursor,NULL,IDC_ARROW
      mov wc.hCursor,        eax
      m2m wc.hIconSm,        NULL
      invoke RegisterClassEx, ADDR wc
      invoke CreateWindowEx,WS_EX_LEFT,
                            ADDR szClassName,
                            ADDR szDisplayName,
                            WS_OVERLAPPEDWINDOW,
                            200,300,420,200,
                            NULL,NULL,
                            hInst,NULL
      mov hWnd,eax
      invoke ShowWindow,hWnd,SW_SHOWNORMAL
```

```
      invoke UpdateWindow,hWnd
    StartLoop:
      invoke GetMessage,ADDR msg,NULL,0,0
      cmp eax, 0
      je ExitLoop
      invoke DispatchMessage,  ADDR msg
      jmp StartLoop
    ExitLoop:
      return msg.wParam
WinMain endp

; оконная функция
WndProc proc hWin   :DWORD,
             uMsg   :DWORD,
             wParam :DWORD,
             lParam :DWORD
    LOCAL count    :DWORD
    LOCAL pic    :DWORD

    Local cordx:DWORD
; локальная координата x
    LOCAL hDC    :DWORD
    LOCAL Ps     :PAINTSTRUCT
    LOCAL tm     :TEXTMETRIC
    LOCAL hPen   :DWORD
    LOCAL hPenp  :DWORD
    szText TT,"My car"
    szText TT1,"Best"
.if uMsg == WM_CREATE
    invoke GetDC,hWin
    mov hDC,eax
    invoke CreatePen,PS_DASHDOT,20,00f59411h
; создание пера
    mov hPen,eax
; сохранение дескриптора пера
    invoke ReleaseDC,hWin, hDC
    return 0
.elseif uMsg==WM_PAINT
      invoke BeginPaint,hWin,ADDR Ps
      mov hDC, eax
      mov count,0
      mov pic,1
      mov cordx,20
      .while count<17  ;цикл вывода изображения
        invoke LoadBitmap,hInstance,pic
        mov hBmp, eax
        inc count
```

```
          add cordx,5
          invoke Paint_Proc,hWin,hDC,cordx
          invoke Pause,12000
          .if pic==1
              mov pic,2
          .else
              mov pic,1
          .endif
        .endw
        mov count,0
        mov pic,3
        .while count<12   ;цикл вывода изображения
          invoke LoadBitmap,hInstance,pic
          mov hBmp, eax
          inc count
          add cordx,10
          invoke Paint_Proc,hWin,hDC,cordx
          invoke Pause,10000
          .if pic==3
              mov pic,4
          .else
              mov pic,3
          .endif
        .endw
        mov count,0
        mov pic,5
        .while count<14                          ;цикл
вывода изображения
          invoke LoadBitmap,hInstance,pic
          mov hBmp, eax
          inc count
          add cordx,15
          invoke Paint_Proc,hWin,hDC,cordx
          invoke Pause,8000
          .if pic==5
              mov pic,6
          .else
              mov pic,5
          .endif
        .endw
        invoke SetBkMode,hDC,1  ;прозрачный фон
        invoke SetTextColor,hDC,00f3434h
; смена цвета шрифта
        invoke TextOut,hDC,140,10,ADDR TT,8
; вывод строки TT
        invoke SelectObject,hDC,hPen
; выбор дескриптора шрифта
```

```
      mov hPenp,eax
; сохранение дескриптора по умолчанию
      invoke SetTextColor,hDC,003252ffh
; смена цвета шрифта
      mov eax,10h
      invoke TextOut,hDC,230,10,ADDR TT1,4
;вывод строки TT1
      invoke SelectObject,hDC,hPenp
      invoke EndPaint,hWin,ADDR Ps
    return 0
.elseif uMsg == WM_DESTROY
        invoke DeleteObject,hPen
;удаление дескриптора пера
        invoke PostQuitMessage,NULL
        return 0
    .endif
    invoke DefWindowProc,hWin,uMsg,wParam,lParam
    ret
WndProc endp
; процедура рисования

Paint_Proc proc hWin:DWORD, hDC:DWORD, cord:DWORD
    LOCAL hOld:DWORD
    LOCAL memDC :DWORD
    invoke CreateCompatibleDC,hDC
    mov memDC, eax
    invoke SelectObject,memDC,hBmp
    mov hOld, eax
; копирование из памяти в окно
    .if cord>=225
       invoke
BitBlt,hDC,cord,50,120,80,memDC,0,0,SRCPAINT
    .else
       invoke BitBlt,hDC,cord,50,120,80,memDC,0,0,SRCCOPY
    .endif
    invoke SelectObject,hDC,hOld
    invoke DeleteDC,memDC
    return 0
Paint_Proc endp
; функция для задержки
Pause proc m:DWORD
  .while m>0
     push m
     .while m>0
       dec m
      .endw
     pop m
```

```
      sub m,1
  .endw
return 0
Pause endp
```

2.7 Упражнения

Написать программу, осуществляющую:

1. Вывод текста с использованием не менее чем двух типов шрифтов
2. Вывод графических примитивов (прямоугольники, эллипсы, дуги и т.п.) с использованием различных перьев и кистей для заливки;
3. Вывод мини-мультфильма по сюжету.

Примеры заданий для мультфильма:

1. Море с видимой линией горизонта и плывущий кораблик.
2. Распускающийся цветок.
3. Кот, бегущий за мышкой.
4. Домик, из трубы которого идет дым.
5. Снеговик, тающий после появления солнышка.
6. С яблони падает яблоко.
7. Солнечное затмение.
8. На ясном небе появляется тучка, идет дождь, затем появляется радуга.
9. Золотая рыбка плавает в аквариуме.
10. В прозрачный стакан капает вода из крана, уровень воды увеличивается.
11. Полет ракеты в космосе.
12. Движущийся автомобиль с мигалкой.
13. Карандаш, рисующий несколько разноцветных линий.
14. Летящая бабочка.
15. Цыпленок, клюющий зерно.
16. Полет Винни-Пуха на воздушном шарике.
17. Прыжок человека с парашютом с самолета.
18. Полет летающей тарелки с изменением ее цвета.
19. Человек, перепрыгивающий через препятствие.
20. Часы с движущимися стрелками.
21. Часы с электронным циферблатом.
22. Часы с кукушкой.
23. Паровоз тянет вагон, из трубы идет дым.
24. Всплывающая подводная лодка.

25. Закат солнца.

2.8 Контрольные вопросы

1. Что такое GDI?
2. Что такое контекст устройства?
3. Какие графические объекты Вы знаете?
4. Что такое регион? Перечислите типы регионов.
5. Как обрабатывается сообщение WM_PAINT?
6. Какое назначение имеют функции GetDC и ReleaseDC?
7. Поясните логику работы с инструментами рисования. Какие функции при этом используются и где вызываются?
8. Поясние назначение функции SelectObject.
9. Какое перо используется по умолчанию?
10. Какие параметры указываются для задания пера?
11. Как запросить для использования дескриптор стандартной кисти?
12. Какие функции используются для создания нестандартных кистей?
13. Перечислите параметры шрифта.
14. Какие шрифты GDI Вы знаете?
15. Что такое базовая линия для физического шрифта?
16. Как изменить цвет символов?
17. Что такое системная палитра?
18. Как формируется логическая палитра?
19. Какие типы растров Вы знаете?
20. Поясните использование ресурсов в приложении.
21. Зачем используется идентификатор ресурса?
22. Что такое совместимый контекст?
23. Как осуществляется вывод DDB-растра?
24. Какие параметры имеет функция BitBlt?
25. Какое назначение имеет функция StretchBlt?

3 РАБОТА С ЭЛЕМЕНТАМИ УПРАВЛЕНИЯ ФОРМ

Для улучшения функциональности приложений и удобства интерфейса на форме обычно размещаются кнопки, списки строк, комбинированные списки, радиокнопки, строки для редактирования информации и т.п. Все эти элементы описываются в Windows как окна предопределенных классов, причем они являются дочерними окнами (стиль WS_CHILD). Каждый элемент управления, как правило, в программе обозначается идентификатором и имеет свой список обрабатываемых сообщений. Идентификатор необходим для различения элементов управления в родительской форме, а точнее, в ее оконной функции.

Элементы форм - создаются с помощью стандартной функции **CreateWindowEx**. Параметры функции:

dwExStyle - расширенный стиль элемента , например может быть WS_EX_NODRAG - элемент невозможно двигать с помощью мыши, WS_EX_NOACTIVATE - элемент не может быть активным, WS_EX_TOPMOST - элемент всегда отображается поверх остальных;

lpClassName - указатель на строку , содержащую название класса элемента , например "EDIT", "BUTTON", "LISTBOX". Именно этот параметр и задаёт тип элемента управления;

lpWindowName - указатель на строку , которая будет надписью;

dwStyle - стиль окна, может быть равен или быть комбинацией следующих констант :

WS_BORDER - элемент с бордюром,

WS_CHILD(или WS_CHILDWINDOW) - элемент является дочерним,

WS_HSCROLL - элемент имеет полосу вертикальной прокрутки,

WS_VSCROLL - элемент имеет полосу горизонтальной прокрутки ,

WS_VISIBLE - элемент изначально видим,

WS_DISABLED - элемент изначально недоступен,

ES_MULTILINE (для EditBox) - элемент является многострочным,

ES_AUTOVSCROLL (для EditBox) - вертикальная автопрокрутка элемента ,

ES_AUTOHSCROLL (для EditBox) - горизонтальная автопрокрутка элемента ,

CS_DBLCLKS - элемент будет воспринимать двойные щелчки мыши.

x, y – координаты верхнего левого угла элемента формы;

Width, Height – ширина и высота элемента управления;

hWndParent - указатель на родительское окно, на которое будет помещён элемент управления;

hMenu - идентификатор дочернего окна, который будет ассоциироваться с элементом;

hInstance – дескриптор экземпляра приложения;

lpParam - указатель на значение, которое будет содержаться в поле lParam для сообщения WM_CREATE, генерируемом после создания элемента.

Функция **CreateWindowEx** возвращает указатель на созданный элемент или NULL в случае невозможности создать элемент формы. Как правило, элементы формы создаются при обработке сообщения WM_CREATE родительской формы .

Взаимодействие с созданным элементом формы осуществляется отправкой и получением сообщений. Для отправки сообщений элементу используется функция **SendMessage**. Функции передаются 4 параметра, первый указывает на элемент формы, второй – код сообщения, третий и четвертый зависят от сообщения. Например, для смены текста на строку в newText используется вызов вида:

```
invoke SendMessage,hElement,WM_SETTEXT,0,ADDR newText
```

Сообщения , которые подходят для всех элементов (в том числе и форм), имеют префикс WM_ (Windows Messages). Однако, для каждого элемента формы существуют специальные сообщения, пригодные только для него, например сообщение LB_GETCOUNT связано со счетчиком строк в ListBox, и для кнопки не имеет смысла. Специальные сообщения для кнопок имеют префикс BM_ (Buttons Messages), для списка строк - LB_ (ListBoxes).

3.1 Кнопки

В простейшем случае кнопка представляет собой прямоугольник, на котором имеется надпись или изображение. Щелчок мышью по кнопке приводит к ее перерисовке с применением тени, чтобы отразить нажатие. Отпускание мыши приводит к восстановлению первоначального вида кнопки. Для того чтобы такая перерисовка стала возможной, от кнопки в родительскую форму посылается сообщение WM_COMMAND. При этом через младшее слово ***wParam*** передается идентификатор кнопки, а через старшее – код действий с кнопкой, например, при нажатии код равен BN_CLICKED.

Для кнопок существуют специализированные стили, названия которых начинаются с префикса BS_. Стиль определяет не только внешний вид кнопки, но и возможность обработки сообщений от клавиатуры и мыши.

Стиль BS_PUSHBUTTON позволяет создать кнопку, которая отправляет сообщение WM_COMMAND родительскому окну, когда пользователь выбирает эту кнопку. Стиль BS_DEFPUSHBUTTON описывает кнопку, которая ведет себя подобно кнопке стиля BS_PUSHBUTTON и имеет жирную черную рамку. Если такая кнопка находится в диалоговом окне, то пользователь может выбрать кнопку, нажав клавишу ENTER, даже тогда, когда кнопка не имеет фокуса ввода. Этот стиль полезен для предоставления пользователю возможности быстро выбрать наиболее подходящую (заданную по умолчанию) опцию.

Стиль BS_GROUPBOX служит для создания прямоугольника, в котором могут быть сгруппированы другие элементы управления. Любой текст, связанный с этим стилем отображается в верхнем левом угле прямоугольника. Кнопка стиля BS_GROUPBOX не предусматривает посылку сообщений родительскому окну и обработку клавиатурных и «мышиных» событий.

Флажок (Check Box) со стилем BS_CHECKBOX представляет собой квадратное окно, работает как переключатель, имеющий 2 состояния – «включено» (в окне рисуется галочка) и «выключено» (в окне ничего не отображено).

Радиокнопка или переключатель со стилем кнопки BS_RADIOBUTTON представляет собой круглое окно, включение которого отражается точкой в окне.

Рассмотрим пример создания кнопки с использованием макроса invoke :

```
.data
    szText  btnClass,"BUTTON"
    szText  text,"MY BUTTON!!!"
    hMybtn  dd 0
...
invoke CreateWindowEx,
    0,ADDR btnClass,ADDR text,
    WS_CHILD or WS_VISIBLE or CS_DBLCLKS,
    100,100,100,20,hWindow,111,
    hInstance,NULL
mov hMybtn,eax
```

Кнопки получают сообщения с помощью функции **SendMessage**.

Для изменения состояния кнопки (нажатое или отжатое) ей передается сообщение BM_SETSTATE. При этом ***wParam*** равен TRUE для перевода кнопки в нажатое состояние, и FALSE для перевода в отжатое. ***lParam*** в обоих случаях равен нулю. Кнопки стиля BS_PUSHBUTTON и BS_DEFPUSHBUTTON при нажатии перерисовываются автоматически. Например:

```
invoke SendMessage,hMyBtn,BM_SETSTATE,TRUE,0
```

Чтобы узнать состояние кнопки, ей передают сообщение BM_GETCHECK с нулевыми параметрами. Возвращаемое значение равно 0 для отжатой кнопки, выключенного переключателя или флажка, 1 - для нажатой кнопки, включенного переключателя или флажка, 2 – для неактивного состояния этих элементов.

Флажки и переключатели имеют стили BS_3STATE, BS_CHECKBOX или BS_RADIOBUTTON, и не перерисовываются автоматически, а для их перерисовки надо послать сообщение BM_SETCHECK. Параметр ***wParam*** указывает, что нужно сделать с переключателем или флажком: 0 – выключить (снять выбор в CheckBox, убрать точку в RadioButton), 1 – включить, 2 – сделать неактивным. Параметр ***lParam*** равен нулю. Например, включить

```
invoke SendMessage,hMyBtn,BM_SETCHECK,1,0
```

Сообщение BM_SETSTYLE устанавливает стиль кнопки, через ***wParam*** передается новый стиль (BS_3STATE, BS_CHECKBOX, BS_AUTOCHECKBOX, BS_AUTO3STATE, BS_RADIOBUTTON, BS_ICON, BS_FLAT т.д.), ***lParam*** равен TRUE, если элемент перерисовывается или FALSE, если не перерисовывается. Например :

```
invoke SendMessage,hMyBtn,BM_SETSTYLE,BS_3STATE,TRUE
```

Слово AUTO в названии стиля означает автоматическую перерисовку объекта при изменении его состояния.

Сообщение BM_SETIMAGE служит для добавления картинки к кнопке, ***wParam*** указывает тип изображения (IMAGE_ICON или IMAGE_BITMAP), ***lParam*** содержит дескриптор изображения. Например:

```
invoke SendMessage,hMyBtn,BM_SETIMAGE,IMAGE_BITMAP,hBmp
```

Возвращаемое значение - указатель на старое изображение на кнопке или NULL , если не было изображения.

Для работы с кнопками можно использовать API-функции для окон, рассмотренные в разделе 1.

3.2 Окно редактирования Edit Box

Edit Box – это прямоугольное окно предопределенного класса «Edit», в котором можно вводить и редактировать текст с клавиатуры. По умолчанию окно редактирования является однострочным, с автоматической горизонтальной прокруткой и выравниванием текста по левой границе.

Для окна редактирования введены дополнительные стили, имеющие префикс ES_:

- ES_AUTOHSCROLL - текст автоматически прокручивается вправо на 10 символов, когда пользователь напечатает символ в конце строки. Когда пользователь нажимает клавишу ENTER, текст прокручивается в начало;
- ES_AUTOVSCROLL - текст автоматически перемещается вверх на одну страницу, когда пользователь нажимает клавишу ENTER на последней строке;
- ES_LEFT – текст выравнивается слева.
- ES_LOWERCASE - символы преобразуются в нижний регистр;
- ES_MULTILINE - окно редактирования является многостроковым. Значение по умолчанию - одностроковое окно редактирования текста.
- ES_NOHIDESEL - выбранный текст инвертируется, даже если панель управления не имеет фокуса;
- ES_NUMBER - в поле редактирования можно ввести только цифры;
- ES_PASSWORD - вместо каждого символа, введенного с клавиатуры в окно редактирования, отображается звездочка (*). Вы можете использовать сообщение EM_SETPASSWORDCHAR, чтобы заменить символ, который отображается;
- ES_READONLY – запрещается ввод или редактирование текста в окне редактирования;
- ES_RIGHT - текст в многострочном окне редактирования выравнивается по правому краю;
- ES_UPPERCASE - все символы преобразуются в символы верхнего регистра.

Сообщения, посылаемые окну редактирования, имеют префикс EM_. Часто употребляемые сообщения приведены в таблице 3.1.

Таблица 3.1 – Сообщения для Edit Box

Код сообщения	Значение wParam	Значение lParam	Назначение
EM_SETSEL	Start	End	Выделение текста с позиции Start до позиции End
EM_GETSEL	адрес Start	адрес End	Получение позиций Start и End выделения
EM_GETLINE	Line	Buf	Копирование строки Line в буфер Buf
WM_CLEAR	0	0	Удаление выделенного текста
WM_CUT	0	0	Удаление выделенного текста и помещение его в буфер обмена Windows
WM_COPY	0	0	Копирование выделенного текста в буфер обмена Windows
WM_PASTE	0	0	Вставка текста из буфера обмена Windows в место текущей позиции курсора
WM_GETTEXT	max	Buf	Копирование не более чем max символов в буфер Buf
WM_SETTEXT	0	адрес Line	Копирование из строки Line в редактор

Например, пусть из строки s1 требуется записать 10 символов в строку редактора с дескриптором hEdit1:

```
invoke SendMessage,hEdit1,WM_GETTEXT,10,ADDR s1
```

Отметим, что для посылки сообщения элементу управления вместо **SendMessage** можно использовать функцию **SendDlgItemMessage**, параметрами которой являются дескриптор родительского окна, идентификатор элемента управления, код сообщения, параметры ***wParam*** и ***lParam***.

От элемента управления EditBox в родительское окно посылаются сообщения WM_COMMAND, при этом младшее слово параметра

wParam содержит идентификатор редактора строки, а старшее – код события с элементом управления. Параметр ***lParam*** содержит дескриптор элемента управления. Примеры кодов событий с элементом EditBox: EN_SETFOCUS/EN_KILLFOCUS – редактор получил/потерял фокус ввода, EN_CHANCE – текст в окне редактирования изменился, EN_ERRSPACE – переполнение буфера редактирования.

Так как текстовый редактор обрабатывает клавиатурные сообщения, то в цикле обработки сообщений обязателен вызов функции **TranslateMessage**.

3.3 Список строк List Box

Элемент управления «список строк» создается на основе предопределенного класса «listbox», отвечающего за формирование окна, в клиентской области которого отображается список, из которого можно осуществлять выбор одного или нескольких элементов. Если размеры окна для списка не позволяют вместить все строки, то создается полоса прокрутки. В списке можно выбрать либо один элемент, либо несколько. Элементы списка могут быть удалены или добавлены.

При создании списка строк стиль указывается с помощью комбинации констант, описывающих стили окон с префиксами WS_ (WS_CHILD и WS_VISIBLE) и стили списков с префиксами LBS_:

- LBS_STANDARD - строки в элементе управления класса LISTBOX сортируются в алфавитном порядке, когда пользователь делает обычный или двойной щелчок на строке, родительскому окну посылается сообщение. Элемент управления класса LISTBOX обладает вертикальной полосой прокрутки и рамками;
- LBS_EXTENDEDSEL - допускается выбор нескольких элементов списка при помощи клавиши <Shift> и мыши или при помощи специальных клавиш;
- LBS_HASSTRINGS - используется для обслуживания памяти и указателей на строки, появляющихся в нестандартном списке. Стиль LBS_HASSTRINGS позволяет приложению использовать сообщение для списка LB_GETTEXT, чтобы получить конкретную строку;
- LBS_MULTICOLUMN – используется для списка из нескольких колонок, между которыми возможна горизонтальная прокрутка;
- LBS_MUPLTIPLESEL - допускается выбор произвольного количества строк, выбранная строка меняется всякий раз, когда пользователь делает обычный или двойной щелчок на строке;
- LBS_NOTIFY - если пользователь сделает обычный или двойной щелчок на строке списка этого стиля, то родительское окно полу-

чит сообщение;

- LBS_OWNERDRAWFIXED- за перерисовку содержимого списка этого стиля отвечает окно-владелец, все элементы списка имеют одинаковую высоту;
- LBS_OWNERDRAWVARIABLE - за перерисовку содержимого списка этого стиля отвечает окно-владелец; в отличие от стиля LBS_OWNERDRAWFIXED, элементы списка могут быть разной высоты.
- LBS_SORT- строки в элементе управления класса LISTBOX сортируются в алфавитном порядке;
- LBS_USETABSTOPS - распознаются и отображаются все символы табуляции при выводе строк;
- LBS_WANTKEYBOARDINPUT - если пользователь нажимает клавишу и список этого стиля обладает фокусом ввода, то владелец списка получает сообщения WM_VKEYTOITEM и WM_CHARTOITEM.

Если стиль списка допускает посылку сообщения WM_COMMAND родительскому окну, то младшее слово параметра ***wParam*** содержит идентификатор списка, старшее – код события со списком. Параметр ***lParam*** содержит дескриптор списка. Примеры кодов событий с элементом ListBox: LBN_DBLCLK – двойной щелчок левой кнопкой мыши по строке списка; LBN_SETFOCUS/LBN_KILLFOCUS – получение/потеря фокуса ввода; LBN_SELCANCEL – отмена выбора в списке; LBN_SELCHANGE – изменился выбор в списке.

Сообщения, посылаемые списку строк, имеют префикс LB_. Часто употребляемые сообщения приведены в таблице 3.2. Отметим, что элементы списка номеруются с нуля.

Таблица 3.2 – Сообщения для List Box

Код сообщения	Значение wParam	Значение lParam	Назначение
LB_ADDSTRING	0	szStr	Добавление в список строки szStr. Если сортировка выключена – то в конец
LB_DELETESTRING	i	0	Удаление i-той строки
LB_GETCOUNT	0	0	Подсчет количества элементов в списке
LB_GETTEXT	i	szStr	Копирование строки с номером i в szStr

Продолжение таблицы 3.2.

Код сообщения	Значение wParam	Значение lParam	Назначение
LB_GETSELITEMS	n	адрес буфера Buf	Запись в буфер Buf индексы выделенных элементов (не более n!) из списка со множественным выбором
LB_FINDSTRING	i	szStr	Поиск строки, начинающейся с szStr в списке, начиная с i+1 элемента. Если i=-1,то с начала списка. **SendMessage** возвращает номер строки или LB_ERR при неудаче
LB_INSERTSTRING	i	szStr	Вставка строки szStr после строки с номером i. Используется для несортированного списка.
LB_RESETCONTENT	0	0	Очистка списка
LB_SELECTSTRING	i	szStr	Поиск аналогично обработке LB_FINDSTRING и выделение найденной строки
LB_SETCURSEL	i	0	Выбор строки с номером i для списка с единичным выбором
LB_SETSEL	TRUE/ FALSE	i	Выбор элемента с номером i в списке со множественным выбором. Если i=-1, то выбираются все элементы списка. Если wParam=TRUE, то выбранные элементы выделяются, FALSE – выбор отменяется.

Продолжение таблицы 3.2.

LB_GETCURSEL	0	0	Для списка с единственным выбором возвращает номер выбранного элемента или LB_ERR
LB_GETSELCOUNT	0	0	Для списка со множественным выбором возвращает количество выбранных элементов

Например, пусть требуется добавить в несортированный список строк строку s1:

```
invoke SendMessage,hEditBox1,LB_ADDSTRING,0,ADDR s1
```

3.4 Комбинированный список Combo Box

Комбинированный список объединяет однострочный текстовый редактор и список строк, создается на основе предопределенного класса «Combobox».

Стиль комбинированного списка, определяемый в параметре ***dwStyle*** функции **CreateWindowEx,** является комбинацией стилей с префиксом WS_ (WS_CHILD, WS_VISIBLE, WS_VSCROLL) и специализированных стилей с префиксами CBS_, перечисленных ниже:

- CBS_SIMPLE - всегда отображается окно со списком, текущий выбор - в поле редактирования текста;
- CBS_AUTOHSCROLL - текст в поле редактирования текста автоматически прокручивается вправо, когда пользователь вводит с клавиатуры символ в конце строки. Если этот стиль не установлен, принимается только текст, который помещается внутри прямоугольной границы поля;
- CBS_DISABLENOSCROLL - в окне со списком вертикальная линейка прокрутки заблокирована, если поле окна содержит недостаточно элементов для прокрутки. Без этого стиля линейка прокрутки скрывается, если окно со списком содержит недостаточно элементов;
- CBS_DROPDOWN – стиль подобен CBS_SIMPLE, за исключением того, что окно со списком не отображается, пока пользователь не выберет значок рядом с полем редактирования текста;
- CBS_DROPDOWNLIST – стиль подобен CBS_DROPDOWN,

за исключением того, что поле редактирования текста заменено статическим текстовым элементом, который отображает текущий выбор в окне со списком.

- CBS_HASSTRINGS - представляемое владельцем комбинированное окно содержит элементы, состоящие из строк. Комбинированное окно поддерживает память и адрес для строк, так что прикладная программа может использовать сообщение CB_GETLBTEXT, чтобы восстановить текст для отдельного элемента;
- CBS_LOWERCASE - преобразовывает в нижний регистр любые символы верхнего регистра, введенные в поле редактирования текста комбинированного окна;
- CBS_NOINTEGRALHEIGHT - определяет, что размер комбинированного окна - это точный размер, определенный прикладной программой, создавшей комбинированное окно. Обычно Windows устанавливает размеры комбинированного окна так, чтобы оно не отображало элементы частично;
- CBS_OEMCONVERT - преобразует текст, введенный в поле редактирования текста комбинированного окна. Текст преобразуется из набора символов Windows в набор символов OEM, а затем обратно в набор Windows. Это гарантирует соответствующее символьное преобразование, когда прикладная программа вызывает функцию CharToOem, чтобы преобразовать строку Windows в комбинированном окне в символы OEM. Этот стиль наиболее полезен для комбинированных окон, которые содержат имена файлов и применяются только в комбинированных окнах, созданных со стилем CBS_SIMPLE или CBS_DROPDOWN;
- CBS_OWNERDRAWFIXED - определяет, что владелец окна со списком ответственен за прорисовку его содержания и что элементы в окне со списком все равной высоты. Окно владельца принимает сообщение WM_MEASUREITEM, когда комбинированное окно создано, а сообщение WM_DRAWITEM, когда внешний вид комбинированного окна изменился;
- CBS_OWNERDRAWVARIABLE – стиль аналогичен CBS_OWNERDRAWFIXED за исключением того, что элементы в окне со списком являются переменными по высоте;
- CBS_SORT – стиль позволяет автоматически сортировать строки, введенные в окно со списком;
- CBS_UPPERCASE – разрешает преобразование символов, введенных в поле редактирования текста комбинированного окна, из нижнего регистра в верхний регистр.

Обмен сообщениями с элементом ComboBox похож на обмен с

элементами ListBox и EditBox. Сообщения, отправляемые ComboBox, имеют префикс CB_, а сообщения для родительского окна, получаемые от ComboBox, имеют префикс CBN_. Примерами сообщений для ComboBox являются CB_GETEDITSEL, CB_SETEDITSEL, CB_ADDSTRING, CB_GETCOUNT, CB_GETCURSEL, CB_GETLBTEXT. От элемента управления класса Combobox" поступают следующие сообщения:

- CBN_CLOSEUP -список стал невидим;
- CBN_DBLCLK – был двойной щелчок левой клавишей мыши по строке списка, имеющего стиль CBS_SIMPLE;
- CBN_DROPDOWN - список стал видимым;
- CBN_EDITCHANGE - пользователь изменил содержимое окна редактирования, причем изменения уже отображены;
- CBN_EDITUPDATE - пользователь изменил содержимое окна редактирования, изменения еще не отображены;
- CBN_ERRSPACE – ошибка;
- CBN_KILLFOCUS - список потерял фокус ввода;
- CBN_SELENDCANCEL - пользователь отменил выбор в списке;
- CBN_SELENDOK - пользователь выбрал строку в списке;
- CBN_SELCHANGE - изменился номер выбранной строки;
- CBN_SETFOCUS - список получил фокус ввода.

3.5 Работа с меню

В большинстве Windows-приложений используется меню. Меню, располагающееся под заголовком окна, называется оконным или главным меню. Плавающее меню может появиться в любом месте после щелчка правой кнопки мыши.

Любое меню состоит из пунктов, обозначаемых словом или графическим изображением. Пункт меню может быть отмечен галочкой, что означает выбор некоторого режима работы приложения. Если после выбора пункта меню на экране должно появиться диалоговое окно, то после названия пункта меню ставится многоточие.

По способу создания различают статическое и динамическое меню. Статическое меню создается заранее и не изменяется в процессе работы программы. Динамическое меню возможно изменять в зависимости от нужд приложения.

Наиболее простым способом создания меню является его описание как ресурса в файле ресурсов с расширением rc. Описание меню имеет следующий вид:

```
идентификатор_меню MENU    [load] [mem]
BEGIN
   . . .
   . . .
END
```

Параметр ***load*** необязателен. Он используется для определения момента загрузки меню в память. Если этот параметр указан как PRELOAD, меню загружается в память сразу после запуска приложения. По умолчанию используется значение LOADONCALL, в этом случае загрузка меню в память происходит только при его отображении.

Параметр ***mem*** также необязателен. Он влияет на тип памяти, выделяемой для хранения шаблона, и может указываться как FIXED (ресурс всегда остается в фиксированной области памяти), MOVEABLE (при необходимости ресурс может перемещаться в памяти, это значение используется по умолчанию) или DISCARDABLE (если ресурс больше не нужен, занимаемая им память может быть использована для других задач). Значение DISCARDABLE может использоваться вместе со значением MOVEABLE.

Между строками BEGIN и END в описании меню располагаются операторы описания строк MENUITEM и операторы описания выпадающих меню POPUP.

Оператор MENUITEM имеет следующий формат:

```
MENUITEM text, id [, param]
```

Параметр ***text*** определяет имя строки меню в двойных кавычках, например, "File". Строка может содержать символы &, \t, \a. Если в строке меню перед буквой стоит знак &, при выводе меню данная буква будет подчеркнута. Например, строка "&File" будет отображаться как "File". Клавиша, соответствующая подчеркнутой букве, может быть использована в комбинации с клавишей <Alt> для ускоренного выбора строки. Для того чтобы записать в строку сам символ &, его следует повторить дважды. Аналогично, для записи в строку меню символа двойной кавычки его также следует повторить дважды.

Символ \t включает в строку меню символ табуляции и может быть использован при выравнивании текста в таблицах. Этот символ обычно используется только во выпадающих и плавающих меню, но не в основном меню приложения, расположенном под заголовком главного окна.

Символ \a выравнивает текст по правой границе выпадающего ме-

ню или полосы меню.

Параметр ***id*** представляет собой целое число, которое должно однозначно идентифицировать строку меню. Приложение получит это число в параметре ***wParam*** сообщения WM_COMMAND, когда будет выбрана данная строка.

Необязательный параметр ***param*** указывается как совокупность атрибутов, разделенных запятой или пробелом. Атрибуты, приведенные ниже, определяют внешний вид и поведение строки меню:

- CHECKED - при выводе меню на экран строка меню отмечается галочкой;
- GRAYED - строка меню отображается серым цветом и находится в неактивном состоянии. Такую строку нельзя выбрать. Этот атрибут несовместим с атрибутом INACTIVE;
- HELP - слева от текста располагается разделитель в виде вертикальной линии;
- INACTIVE - строка меню отображается в нормальном виде (не серым цветом), но находится в неактивном состоянии. Этот атрибут несовместим с атрибутом GRAYED;
- MENUBREAK - если описывается меню верхнего уровня, элемент меню выводится с новой строки. Если описывается выпадающее меню, элемент меню выводится в новом столбце;
- MENUBARBREAK - аналогично атрибуту MENUBREAK, но дополнительно новый столбец отделяется вертикальной линией (используется при создании выпадающих меню).

Для описания выпадающих меню используется оператор POPUP:

```
POPUP text [, param]
BEGIN
  . . .
  . . .
END
```

Между строками BEGIN и END в описании выпадающего меню располагаются операторы описания строк MENUITEM и операторы описания вложенных меню POPUP. Параметры ***text*** и ***param*** указываются так же, как и для оператора MENUITEM.

Для того чтобы создать в меню горизонтальную разделительную линию, используется специальный вид оператора MENUITEM:

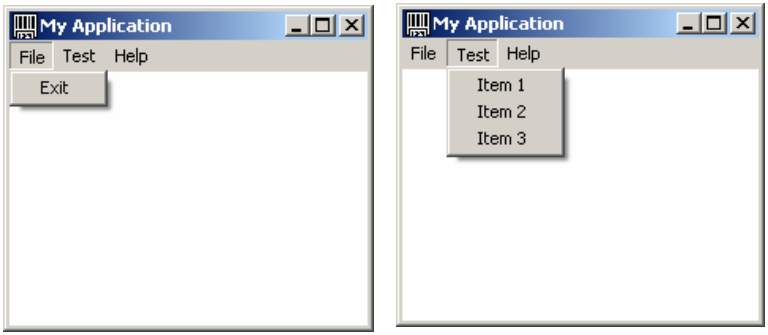
```
MENUITEM SEPARATOR
```

Поясним сказанное выше на простом примере. Пусть необходимо

создать главное меню, включающее пункты File, Test и Help, причем пунктам File и Help соответствует выпадающее меню из одного пункта, пункту Test – выпадающее меню из трех пунктов. Такое меню будет описано в файле ресурсов следующим образом:

```
MYAPP MENU DISCARDABLE
BEGIN
    POPUP "File"
    BEGIN
        MENUITEM "Exit", 101
    END
    POPUP "Test"
    BEGIN
        MENUITEM "Item 1", 201
        MENUITEM "Item 2", 202
        MENUITEM "Item 3", 203
    END
    POPUP "Help"
    BEGIN
        MENUITEM "About My Application...", 301
    END
END
```

При выборе пункта «File» окно приложения имеет вид, представленный на рисунке 3.1,а, а при выборе пункта «Test» - вид, представленный на рисунке 3.1,б.



а б

Рисунок 3.1

Следующий этап после создания ресурса меню - подключение меню к окну приложения.

Подключить меню можно на этапе регистрации класса окна или для отдельного окна при его создании функцией **CreateWindow(Ex)**.

Если при регистрации класса окна в поле ***lpszMenuName*** структуры типа WNDCLASS указать адрес текстовой строки, содержащей имя шаблона меню в файле ресурсов, все перекрывающиеся и временные окна, создаваемые на базе этого класса, будут иметь меню, определенное данным шаблоном. Дочерние окна (child window) не могут иметь меню. Например, пусть в файле описания ресурсов шаблон меню определен под именем APP_MENU:

```
APP_MENU MENU
BEGIN
....
END
```

В этом случае для подключения меню при регистрации класса вы должны записать адрес текстовой строки "APP_MENU" в поле ***lpszMenuName*** структуры ***wc***, имеющей тип WNDCLASS:

```
wc.lpszMenuName = "APP_MENU";
```

Когда для класса окна определено меню, все перекрывающиеся и временные окна, создаваемые на базе этого класса, будут иметь меню, если при создании окна функцией **CreateWindow** указан идентификатор меню, равный нулю.

Для подключения меню, отличного от указанного в классе окна, необходимо задать идентификатор нужного меню при создании окна функцией **CreateWindow** через 9 параметр этой функции дескриптор меню ***hMenu***.

Значение параметра идентификатора меню может быть получено, например, от функции **LoadMenu** , определенной в программном интерфейсе Windows. Параметры **LoadMenu:**

- дескриптор текущей копии приложения, полученный через соответствующий параметр функции **WinMain**;

- указатель ***lpszMenuName*** на строку символов, заканчивающуюся двоичным нулем и содержащую имя загружаемого шаблона меню.

Перед завершением своей работы приложение должно уничтожить загруженное меню функцией **DestroyMenu**, которой передается его дескриптор.

Для динамического создания и изменения меню используются функции **CreateMenu, AppendMenu, LoadMenuIndirect, InsertMenu, RemoveMenu, SetMenuItemInfo, CheckMenuItem, EnableMenuItem,**

ModifyMenu.

Меню посылает в оконную функцию создавшего его окна следующие сообщения. Сообщение WM_INITMENU посылается перед отображением меню и может быть использовано для инициализации. Сообщение WM_COMMAND посылается после того, как пользователь выберет одну из активных строк меню. Системное меню посылает в окно приложения сообщение WM_SYSCOMMAND, которое обычно не обрабатывается приложением (передается функции **DefWindowProc**). В процессе выбора строки из меню, когда курсор перемещается по строкам меню, функция окна, создавшего меню, получает сообщение WM_MENUSELECT. Перед инициализацией выпадающего меню функция окна получает сообщение WM_INITMENUPOPUP.

Из всех этих сообщений наибольший интерес представляют сообщения WM_INITMENU, WM_INITMENUPOPUP, WM_COMMAND, WM_SYSCOMMAND. Рассмотрим их более подробно.

Сообщение WM_INITMENU посылается окну, создавшему меню, в момент отображения меню. Это происходит при нажатии на строку в полосе меню или активизации выпадающего меню при помощи клавиатуры. Вместе с этим сообщением в параметре ***wParam*** передается идентификатор активизируемого меню. Обработка WM_INITMENU может заключаться в активизации или деактивизации строк меню, изменении состояния строк (отмеченное галочкой или не отмеченное) и т. п. с помощью вышеперечисленных функций работы с динамическим меню.

Сообщение WM_INITMENUPOPUP посылается окну, когда Windows готова отобразить выпадающее меню. Младшее слово параметра ***lParam*** содержит порядковый номер выпадающего меню в меню верхнего уровня, старшее слово содержит 1 для системного меню или 0 для обычного меню. Это сообщение можно использовать для активизации или блокирования отдельных строк выпадающего меню.

Сообщение WM_MENUSELECT генерируется в процессе перемещения курсора по строкам меню, его ценность заключается в отображении текущих действий пользователя. Младшее слово ***wParam*** содержит идентификатор пункта меню. Старшее слово ***wParam*** содержит состояние пункта меню:

MF_CHECKED – отмечен;

MF_DISABLED – заблокирован;

MF_GRAYED – недоступен;

MF_HILITE – высвечен;

MF_MOUSESELECT – выбран мышью;

MF_POPUP – открывается выпадающее меню.

Сообщение WM_COMMAND посылается при выборе пункта меню. Параметр ***wParam*** содержит идентификатор пункта меню, определенный в шаблоне меню. В оконной функции, обрабатывающей сообщения от меню, значения параметра wParam проверяются и выполняются соответствующие действия.

Сообщение WM_SYSCOMMAND приходит в функцию окна приложения, когда пользователь выбирает строку из системного меню. Параметр ***wParam***, как и для сообщения WM_COMMAND, содержит идентификатор строки меню, в данном случае, идентификатор строки системного меню.

Рассмотрим пример обработки выбора пунктов меню в оконной функции. Пусть в файле menu.rc описано меню, состоящее из 4 пунктов, каждому из которых назначен свой идентификатор:

```
#define IDM_TEST 100
#define IDM_HELLO 200
#define IDM_GOODBYE 300
#define IDM_EXIT 400
700 MENU DISCARDABLE
BEGIN
   POPUP "&Show"
   BEGIN
      MENUITEM "Test", IDM_TEST
      MENUITEM "Hello",  IDM_HELLO
      MENUITEM "Bye",  IDM_GOODBYE
      MENUITEM "Exit", IDM_EXIT
   END
END
```

Обработка выбора этих пунктов, за исключением пункта EXIT, заключается в выводе текстовых сообщений, описанных как строки, завершающиеся двоичными нулями:

```
szText Test_string,"Test menu item"
szText Hello_string,"Hello!"
szText Goodbye_string,"Bye!"
```

Оконная функция имеет вид:

```
WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM,
             lParam:LPARAM
      .IF uMsg==WM_DESTROY
          invoke PostQuitMessage,NULL
```

```
    .ELSEIF uMsg==WM_COMMAND
        mov eax,wParam
        .IF eax==IDM_TEST
            invoke MessageBox, NULL,
                   ADDR Test_string,
                   OFFSET AppName, MB_OK
        .ELSEIF eax==IDM_HELLO
            invoke MessageBox, NULL,
                   ADDR Hello_string,
                   OFFSET AppName,MB_OK
        .ELSEIF eax==IDM_GOODBYE
            invoke MessageBox,NULL,
                   ADDR Goodbye_string,
                   OFFSET AppName, MB_OK
        .ELSE
            invoke DestroyWindow,hWnd
        .ENDIF
    .ELSE
        invoke DefWindowProc, hWnd, uMsg, wParam,
              lParam
        ret
    .ENDIF
    xor eax,eax
    ret
WndProc endp
```

Для работы с плавающим меню используется функция **TrackPopupMenu**, которая выводит плавающее меню на экран и создает собственный цикл обработки сообщений, завершающийся после выбора строки из этого меню. Управление не возвращается до тех пор, пока не будет выбрана строка или пока не будет получен отказ от выбора. Рассмотрим параметры функции:

- дескриптор плавающего меню. Формируется функцией **CreatePopupMenu** или **GetSubMenu**;

- целое беззнаковое число, биты которого задают параметры для меню (размещение меню по горизонтали и вертикали относительно соответственно координат x и y, задаваемых 3 и 4 параметрами функции **TrackPopupMenu,** наличие или отсутствие посылки сообщений при выборе строк меню, по правой или левой кнопке мыши выводится плавающее меню**);**

- x и **y**-координаты относительно левого верхнего угла экрана;

- зарезервированный параметр, равный нулю при вызове;

- дескриптор окна, которому адресованы сообщения от меню;

- указатель на прямоугольную область экрана, в пределах которой

щелчки по пунктам меню будут обработаны. Если щелчок будет вне этой области, то плавающее меню исчезнет.

Функция **TrackPopupMenu** возвращает идентификатор выбранной строки или нулевое значение, если ничего не было выбрано.

3.6 Пример программы по работе с элементами управления

Рассмотрим приложение, иллюстрирующее ряд вышеперечисленных возможностей элементов управления. На форме разместим объекты следующих типов: List Box, Edit Box и две кнопки, одна из которых предназначена для перемещения текста из строки редактирования в список строк, а другая - для перемещения выделенной строки из списка в строку редактирования. Двойной щелчок по строке списка или нажатие клавиши «Enter» приводит к выводу окна с сообщением, текст которого берется из текущего элемента списка строк. Кроме того, создадим на форме надпись, являющуюся статическим элементом управления и предствляющую собой окно предопределенного класса «STATIC».

Текст программы приведен ниже.

```
.386
.model flat,stdcall
option casemap:none
; Подключение файлов и библиотек
include c:\masm32\include\windows.inc
include c:\masm32\include\user32.inc
include c:\masm32\include\kernel32.inc
includelib c:\masm32\lib\user32.lib
includelib c:\masm32\lib\kernel32.lib

; прототипы функций
WinMain proto:DWORD,:DWORD,:DWORD
WinProc proto:DWORD,:DWORD,:DWORD,:DWORD
ListBoxProc proto :DWORD,:DWORD,:DWORD,:DWORD
; идентификаторы кнопок
bn1_id       equ    501
bn2_id       equ    502
; константы
.const
szListBox    db     "LISTBOX",0
szButton     db     "BUTTON",0
szEdit       db     "EDIT",0
szStatic     db     "STATIC",0
```

```
szBn1         db     "<",0
szBn2         db     ">",0
szTip         db     "Введите строку в текстовое поле и
нажмите '<' для добавления в список.",0Dh,0Ah,0Dh,0Ah,
"Выделите элемент списка и нажмите '>' для чтения",
"строки.",0Dh,0Ah,0Dh,0Ah,"Двойной щелчок левой", "кноп-
ки мыши по элементу списка или нажатие Enter","вызывает
окно с сообщением.",0
szClassName  db     "Class1",0
szWindowName db     "Window1",0
.data
; неинициализированные переменные:
hInstance HINSTANCE ?
; идентификатор нашего процесса
CommandLine LPSTR ? ; командная строка
hEdit HWND ? ; дескриптор окна текстового поля
hList HWND ? ; дескриптор окна списка
lpLstBox DWORD ?
; указатель на предыдущую функцию ListBoxProc
szBuffer db 128 dup(?)     ; текстовый буфер
.code
start:
  invoke GetModuleHandle, NULL
; получение идентификатора процесса
  mov hInstance, eax
  invoke GetCommandLine
; получение указателя командной строки
  mov CommandLine, eax
  invoke WinMain,hInstance,NULL,CommandLine
; вызов основной функции
  invoke ExitProcess,eax
; выход из программы (с кодом возврата в eax)
WinMain proc hInst:HINSTANCE, hPrevInst:HINSTANCE,
             CmdLine:LPSTR
; локальные переменные:
  LOCAL wc:WNDCLASSEX      ; класс окна
  LOCAL msg:MSG            ; структура сообщения
  LOCAL hwnd:HWND          ; дескриптор окна
;=================================================
; Заполнение и регистрация класса окна WNDCLASSEX
;=================================================
  mov wc.cbSize,SIZEOF WNDCLASSEX
; размер структуры 4*12 байтов
  mov wc.style, CS_HREDRAW or CS_VREDRAW`
; стиль окна, предусматривающий перерисовку
; при вертикальном и горизонтальном движении
  mov wc.lpfnWndProc, OFFSET WndProc
```

```
; функция-обработчик событий окна
  mov wc.cbClsExtra,NULL
; число дополнительных байтов
  mov wc.cbWndExtra,NULL
; число дополнительных байтов
  push hInst
  pop wc.hInstance
; идентификатор нашего процесса
  mov wc.hbrBackground,COLOR_BTNFACE+1
; идентификатор кисти (или цвет фона+1)
  mov wc.lpszMenuName,NULL
; ресурс с основным меню
  mov wc.lpszClassName,OFFSET szClassName
; имя класса
  invoke LoadIcon,NULL,IDI_APPLICATION
  mov wc.hIcon,eax
; идентификатор иконки
  mov wc.hIconSm,NULL
; идентификатор маленькой иконки (должен быть 0)
  invoke LoadCursor,NULL,IDC_ARROW
  mov wc.hCursor,eax
; идентификатор курсора
  invoke RegisterClassEx, addr wc
; регистрация класса
;===============
; Создание окна
;===============
; Параметры функции CreateWindowEx:
; 1.  дополнительный стиль (0 - по умолчанию)
; 2.  наш зарегистрированный класс
; 3.  заголовок окна
; 4.  стиль окна (стандартное изменяемое по размеру окно
с системными кнопками)
; 5.  x-координата (CW_USEDEFAULT - по умолчанию)
; 6.  y-координата (по умолчанию)
; 7.  ширина окна
; 8.  высота окна
; 9.  идентификатор окна-предка
; 10. идентификатор меню или окна-потомка
; 11. идентификатор процесса, который будет получать
; сообщения от окна
; 12. адрес структуры CREATESTRUCT (не используется)
  invoke CreateWindowEx, 0, ADDR szClassName,
         ADDR szWindowName, WS_OVERLAPPEDWINDOW ,
         CW_USEDEFAULT, CW_USEDEFAULT, 500, 400, 0, 0,
         hInst, 0
  mov hwnd,eax
```

```
;==================
; Отображение окна
;==================
  invoke ShowWindow,hwnd,SW_SHOWNORMAL
; показать окно
  invoke UpdateWindow,hwnd
; и послать ему сообщение WM_PAINT
;=========================
; Цикл обработки сообщений
;=========================
message_loop:
  invoke GetMessage,ADDR msg,0,0,0
; получить сообщение от окна
  test eax,eax
; если получено WM_QUIT, GetMessage вернет ноль
  jz exit_msg_loop
; выйти
  invoke TranslateMessage,ADDR msg
; иначе - преобразовать сообщения типа WM_KEYUP
; в сообщения типа WM_CHAR
  invoke DispatchMessage,ADDR msg
; и послать их оконной функции
  jmp short message_loop  ; продолжить цикл
exit_msg_loop:
  mov eax,msg.wParam                    ; код возврата
  ret
WinMain endp

WndProc proc hWnd:HWND, uMsg:UINT, wParam:WPARAM,
             lParam:LPARAM
  .IF uMsg == WM_DESTROY
    invoke PostQuitMessage,0     ; послать WM_QUIT
  .ELSEIF uMsg == WM_CREATE
;=========================
; Создание списка ListBox
;=========================
; Параметры при создании списка:
; 1.  дополнительный стиль (0 - по умолчанию)
; 2.  класс LISTBOX (системный класс)
; 3.  пустой заголовок окна
; 4.  стиль окна: WS_VISIBLE - сделать ListBox види
; мым, WS_VSCROLL - использовать вертикальную полосу
; прокрутки, WS_BORDER - линия контура, WS_CHILD -
; разместить ListBox внутри формы, LBS_HASSTRINGS -
; элементы ListBox являются строковыми,
; LBS_NOINTEGRALHEIGHT - не округлять размер ListBox
; до целого числа строк (обычно Windows устанавливает
```

```
; размеры списка так, чтобы он не обрезал часть
; не вошедших по высоте окна списка строк,
; LBS_DISABLENOSCROLL - всегда отображать полосы
; прокрутки
; 5.  x-координата
; 6.  y-координата
; 7.  ширина
; 8.  высота
; 9.  идентификатор окна-предка
; 10. идентификатор меню или окна-потомка
; 11. идентификатор процесса, который будет получать
; сообщения от окна (наш процесс)
; 12. адрес структуры CREATESTRUCT (не используется)
  invoke CreateWindowEx,0,ADDR szListBox,0,
         WS_VISIBLE or WS_VSCROLL or  \
         WS_BORDER or WS_CHILD or \
         LBS_HASSTRINGS or LBS_NOINTEGRALHEIGHT or \
         LBS_DISABLENOSCROLL,10,10,250,350,hWnd,
         0,hInstance,0
  mov hList,eax     ; дескриптор окна ListBox
; Изменение атрибутов окна ListBox: установка оконной
; функции ListBoxProc. Эта функция аналогична WndProc.
; Она необходима для обработки событий нажатия клавиш
; в окне ListBox.
  invoke SetWindowLong,hList,GWL_WNDPROC,ListBoxProc
  mov lpLstBox, eax ; запоминание старой функции
;=================
; Создание кнопок
;=================
; Кнопка 1
  invoke CreateWindowEx,0, ADDR szButton,ADDR szBn1,
         WS_CHILD or WS_VISIBLE,70,10,30, 30,hWnd,
         bn1_id, hInstance,0
; Кнопка 2
  invoke CreateWindowEx,0, ADDR szButton,ADDR szBn2,
         WS_CHILD or WS_VISIBLE,270,50,30,30,hWnd,
         bn2_id, hInstance,0
;==============================
; Создание текстового поля Edit
;==============================
; ES_AUTOHSCROLL - автоматический сдвиг символов влево
; при нехватке места для текста,
; ES_NOHIDESEL - отключает скрытие выделения при
; переходе к другому элементу управления
  invoke CreateWindowEx,0,ADDR szEdit,0,
        WS_BORDER or WS_VISIBLE or WS_CHILDWINDOW or \
        ES_AUTOHSCROLL or ES_NOHIDESEL,310,10,170,70,
```

```
        hWnd, 0,hInstance,0
  mov hEdit,eax     ; дескриптор окна Edit
;================================================
; Создание неизменяемого поля с надписью Static
;================================================
  invoke CreateWindowEx,0, ADDR szStatic,ADDR szTip,
         WS_CHILD or WS_BORDER or WS_VISIBLE,
         270,100,210,250,hWnd,0,hInstance,0
;=========================
; Обработка нажатия кнопок
;=========================
  .ELSEIF uMsg == WM_COMMAND
    .IF wParam == bn1_id
; нажата кнопка '<' - перемещение в ListBox
; копирование текста окна Edit в буфер
; с максимальной длиной текста 127 байт
       invoke GetWindowText,hEdit,ADDR szBuffer,127
; отправка сообщения LB_ADDSTRING о добавлении строки
; окну списка ListBox
       invoke SendMessage,hList,LB_ADDSTRING,0,
              ADDR szBuffer
    .ELSEIF wParam == bn2_id     ; нажата кнопка '>'
; отправка сообщения LB_GETCURSEL о получении индекса
; выделенной строки окну списка ListBox
      invoke SendMessage,hList,LB_GETCURSEL,0,0
      push eax
; отправка сообщения LB_GETTEXT о получении указанной
; строки (находится в eax) окну списка ListBox
      invoke SendMessage,hList,LB_GETTEXT,eax,
             ADDR szBuffer
      pop eax
; отправка сообщения LB_DELETESTRING об удалении
; указанной строки (находится в eax) окну списка
; ListBox
      invoke SendMessage,hList,LB_DELETESTRING,eax,0
; изменяет текст указанного окна
      invoke SetWindowText,hEdit,ADDR szBuffer
    .ENDIF
  .ELSE
; для остальных сообщений - обработчик по умолчанию
  invoke     DefWindowProc,hWnd,uMsg,wParam,lParam
  ret
  .ENDIF
  xor eax,eax
  ret
WndProc endp
ListBoxProc proc hCtl:HWND, uMsg:UINT, wParam:WPARAM,
```

```
                    lParam:LPARAM
  .IF uMsg == WM_LBUTTONDBLCLK
; если произведен двойной щелчок
       jmp DoIt
  .ELSEIF uMsg == WM_CHAR
; или нажата клавиша Enter
     .IF wParam == 13
          jmp DoIt  ; то вывести сообщение
     .ENDIF
  .ENDIF
  jmp EndDo
DoIt:
  invoke SendMessage,hCtl,LB_GETCURSEL,0,0              ;
получение текста выделенной строки
  invoke SendMessage,hCtl,LB_GETTEXT,eax,ADDR szBuffer
; и вывод ее в окне
  invoke MessageBox,hCtl,ADDR szBuffer,
         ADDR szListBox,MB_OK    ; вызов MessageBox
EndDo:
; Вызов старой функции ListBoxProc для обработки
; остальных сообщений
  invoke CallWindowProc,lpLstBox,hCtl,uMsg,wParam,
         lParam
  ret
ListBoxProc endp
end start
```

Результаты тестирования программы приведены на рисунках 3.2-3.4.

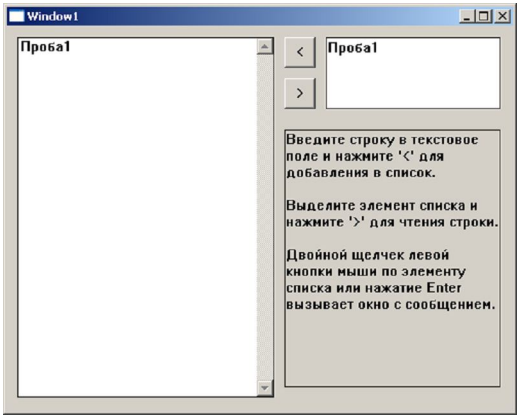


Рисунок 3.2 - Вставка строки «Проба1» при нажатии кнопки '<'.

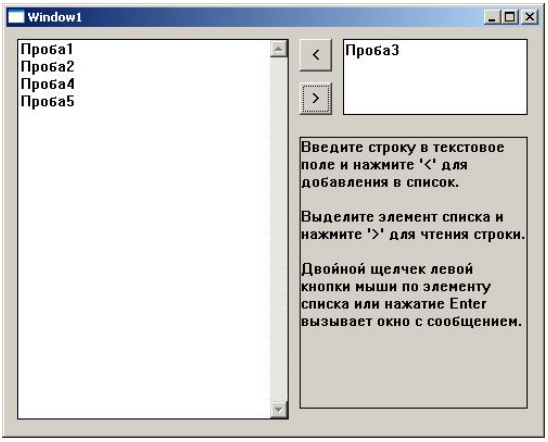


Рисунок 3.3 - Считывание активной строки при нажатии кнопки '>'.

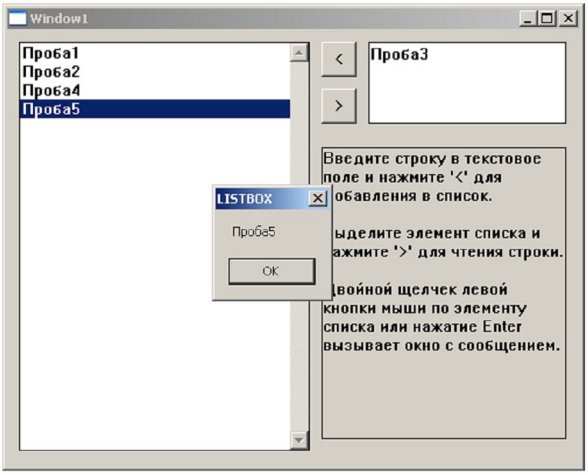


Рисунок 3.4 – Вывод сообщения с текстом выбранного элемента списка при нажатии кнопки Enter или двойном щелчке мыши

3.7 Упражнения

Варианты заданий:

1. Реализовать занесение строк из комбинированного списка в список строк при двойном щелчке левой кнопки по элементу из ком-

бмнированного списка.

2. В клиентской области окна приложения создать 2 списка, первый из которых содержит числа от 1 до 5, второй – числа от 200 до 205. При нажатии кнопки «Добавить» в список с фокусом ввода добавить очередное число.

3. В клиентской области окна приложения отобразить список строк, состоящий из чисел от 1 до 4, и 2 кнопки - «Добавить», «Удалить». Кнопка «Добавить» добавляет в список новый элемент, на единицу больший последнего элемента списка. Сообщить пользователю о действиях над этими кнопками.

4. В клиентской области окна приложения отобразить список строк строк, состоящий из чисел от 1 до 4, и 2 кнопки -«Удалить» и «Добавить измененный элемент». Кнопка «Добавить измененный элемент» добавляет в список новый элемент, который представляет собой конкатенацию текущего элемента и дополнительной буквы «А». Сообщить пользователю о действиях над этими кнопками.

5. В окне приложения отобразить однострочный редактор, под ним 2 списка и еще ниже – 2 кнопки с надписями «Поместить в первый список» и «Поместить во второй список», при нажатии на которые текст из редактора помещается в один из списков.

6. Окно содержит однострочный текстовый редактор с надписью «Введите фамилию», группу переключателей «Пол» из двух пунктов («Мужчина» и «Женщина»), кнопку «Занести» и список строк. При нажатии на кнопку «Занести» конкатенацию содержимого строки ввода и первой буквы пола добавить в список и очистить строку ввода.

7. В центре экрана отобразить окно, содержащее фрагмент текста и кнопки «Выход» и «Далее». При нажатии кнопки «Выход» завершить работу, а кнопки «Далее» - текст в окне заменить другим фрагментом, а кнопки заменить кнопками «Назад» и «Выход». При нажатии кнопки «Назад» вернуться к исходному состоянию.

8. На форме разместить 4 элемента Check Box с надписями «Сильный», «Смелый», «Добрый», «Умный». При нажатии на кнопку «Добавить» выбранные качества отображаются с использованием MessageBox.

9. В окне отобразить два однострочных текстовых редактора и кнопку «Обмен». При нажатии кнопки происходит обмен текстовых строк в редакторах и выводится соответствующее сообщение.

10. В нижней части клиентской области окна отобразить кнопку с надписью «Показать», в верхней части окна – статическую надпись «Введите текст». После нажатия на кнопку в средней части окна отображается многострочный редактор с исходным текстом «Нажмите

правую клавишу мыши». После нажатия правой клавиши мыши скрыть редактор.

11. В клиентской области окна разместить комбинированный список. При выборе элемента списка показывать сообщение, текст которого – выбранный элемент. При нажатии левой клавиши мыши в верхней половине клиентской области инициализировать список новым набором значений. Число наборов – не менее трех, текущий набор меняется циклически.

12. На форме разместить список строк, однострочный редактор и кнопка «Добавить». При нажатии на кнопку вставить строку из редактора в середину списка.

13. На форме разместить список строк, однострочный редактор и кнопка «Найти». При нажатии на кнопку найти строку из редактора в списке строк. Если найти не удалось, то сообщить об этом.

14. В верхней части клиентской области окна находится надпись, показывающая номер текущей строки, выбранной в списке строк.

15. Создать однострочный редактор, список и кнопку «Изменить». При нажатии на кнопку «Изменить» содержимое выбранной строки списка передать в редактор и удалить из списка.

16. Создать 2 списка строк, однострочный редактор, и кнопку «Найти», при нажатии на которую содержимое редактора (если оно не пустое) использовать в качестве начальных букв поиска строки в списках. Сообщить, удалось ли найти строку и в каком списке .

17. В окне разместить группу из 3 переключателей, группу из 4 флажков, кнопки «Состояние» и «Выход», причем кнопка «Состояние» выбирается по умолчанию. При нажатии кнопки «Состояние» сообщить о состоянии переключателей и флажков. Если нажата кнопка «Выход», то завершить работу.

18. В клиентской области окна разместить список строк, группу флажков, способных находиться в трех состояниях, с надписями «Группа», «Факультет» и кнопку «Добавить». После нажатия кнопки «Добавить» добавить в конец списка строк в зависимости от установленного флажка строку, являющуюся конкатенацией надписей в выбранных флажках и порядкового номера добавляемой строки в List Box.

19. В клиентской области окна разместить три списка строк. В первый занести названия изучаемых в семестре дисциплин, во второй – форму контроля (зачет или экзамен). При нажатии на кнопку «Сессия» занести в третий список те дисциплины, по которым в сессию предусмотрен экзамен.

20. В верхней части формы разместить текст «Анкета студента»,

ниже – поля ввода фамилии, имени, группы, и группы переключателей с номером курса. Окно содержит кнопки «Записать» и «Очистить». При нажатии кнопки «Записать» выдается сообщение «Запись осуществлена», при нажатии кнопки «Очистить» все поля чистятся.

21. Отобразить комбинированный список, содержащий названия специальностей, однострочный редактор с надписью «Фамилия» и кнопку «Записать». При нажатии кнопки конкатенация строки из редактора и названия выбранной специальности записывается в список строк.

22. Комбинированный список заполнить произвольными различными строками. Рядом расположить кнопку «Вниз». Если выбрать строку и нажать кнопку, то эта строка должна переместиться вниз на одну позицию в списке. Если эта строка уже последняя в списке, то сообщить об этом и заблокировать кнопку.

23. Создать строку ввода с числом 100 и кнопки «Увеличить» и «Уменьшить». При нажатии на кнопку «Увеличить» число в строке ввода увеличивается на единицу, а при нажатии на кнопку «Уменьшить» – уменьшается на единицу. При нажатии кнопки «Выход» - завершить работу.

24. Создать два списка строк соответственно с надписями «Фрукты» и «Овощи», группу из двух переключателей и кнопку «Выбрать». После нажатия на кнопку выдается сообщение о первом элементе того списка, который отмечен переключателем.

25. Создать два списка, между ними расположить кнопки «Переместить>>» и «<<Переместить”. Левый список должен быть изначально заполнен списком строк. Если выбрать строку и нажать одну из кнопок, то выбранная строка должна переместиться слева направо или наоборот. Списки должны быть отсортированы.

3.8 Контрольные вопросы

1. Какие предопределенные классы для элементов управления Вы знаете?
2. Зачем для элемента управления используется идентификатор?
3. Как создаются элементы управления?
4. Перечислите параметры функции CreateWindowEx.
5. Как осуществляется взаимодействие с элементом управления?
6. Как по префиксу сообщения определить, к какому элементу управления оно относится?
7. Какие особенности имеет элемент «Кнопка»?
8. Какие стили кнопок Вы знаете?

9. Как перерисовать флажок или переключатель с изменившимся состоянием?
10. Какие типы окон редактирования Вы знаете?
11. Какие стили имеет окно редактирования?
12. Какие сообщения посылает EditBox в родительское окно?
13. Перечислите сообщения, передаваемые EditBox из родительского окна.
14. Перечислите возможные действия со списком строк.
15. Какие стили для списка строк Вы знаете?
16. Какие сообщения передаются элементу ListBox?
17. Как нумеруются элементы списка строк?
18. Что такое комбинированный список?
19. Какие стили комбинированного списка Вы знаете?
20. Какие сообщения посылаются ComboBox?
21. Как описать статическое меню в файле ресурсов?
22. Зачем используется знак “&” при описании элемента меню?
23. Как подключается меню в приложении?
24. Какие функции используются для работы с динамическим меню?
25. Что такое плавающее меню? Какая функция работает с плавающим меню?

4 ДИНАМИЧЕСКИЕ БИБЛИОТЕКИ

4.1 Общие сведения

Библиотеки динамической компоновки (dynamic link libraries -DLL) являются исполняемыми файлами особого формата, которые содержат функции, данные или ресурсы, доступные для других приложений.

Особый формат модулей DLL предполагает наличие в них разделов импорта и экспорта. Раздел экспорта указывает те идентификаторы объектов (функций, классов, переменных), доступ к которым разрешен для клиентов.

Подавляющее большинство DLL (за исключением DLL, содержащих только ресурсы) импортирует функции из системных DLL – kernel32.dll, user32.dll, gdi32.dll и других библиотек.

Применение DLL может дать ряд преимуществ:

- *Расширение функциональности приложения.* DLL можно загружать в адресное пространство процесса на этапе выполнения, что позволит программе, определив, какие действия от нее требуются, подгружать нужный код.
- *Более простое управление проектом.* Использование DLL упрощает отладку, тестирование и сопровождение проекта.
- *Экономия памяти.* Если одну и ту же DLL используют несколько приложений, то в оперативной памяти хранится только один ее экземпляр, доступный этим приложениям.
- *Разделение ресурсов.* DLL могут содержать такие ресурсы, как строки, растровые изображения, шаблоны диалоговых окон. Этими ресурсами может воспользоваться любое приложение.
- *Возможность использования разных языков программирования*.

Исполняемый код в DLL не предполагает автономного использования. Содержимое каждого DLL–файла загружается приложением и проецируется на адресное пространство вызывающего процесса. Это достигается либо за счет неявного связывания при загрузке, либо за счет явного связывания в период выполнения.

Важно понимать, что процесс, загрузивший DLL, получает собственную копию глобальных данных, используемых этой библиотекой. Это защищает DLL от ошибок приложений, а процессы, использующие DLL, от взаимного влияния друг на друга.

4.2 Вызов функций из DLL

Существует три способа загрузки DLL:
а) неявная;
б) явная;
в) отложенная.

Рассмотрим неявную загрузку DLL. Для построения приложения, рассчитанного на неявную загрузку DLL, необходимо иметь:

- Библиотечный включаемый файл с описаниями используемых объектов из DLL (прототипы функций, объявления классов и типов). Этот файл используется компилятором.
- LIB–файл со списком импортируемых идентификаторов. Этот файл нужно добавить в настройки проекта (в список библиотек, используемых компоновщиком).

Компиляция проекта осуществляется обычным образом. Используя объектные модули и LIB – файл, а также учитывая ссылки на импортируемые идентификаторы, компоновщик (линкер, редактор связей) формирует загрузочный EXE – модуль. В этом модуле компоновщик помещает также раздел импорта, где перечисляются имена всех необходимых DLL-модулей. Для каждой DLL в разделе импорта указывается, на какие имена функций и переменных встречаются ссылки в коде исполняемого файла. Эти сведения будет использовать загрузчик операционной системы.

Что же происходит на этапе выполнения клиентского приложения? После запуска EXE-модуля загрузчик операционной системы выполняет следующие операции:

1. Создает виртуальное адресное пространство для нового процесса и проецирует на него исполняемый модуль;
2. Анализирует раздел импорта, определяя все необходимые DLL-модули и тоже проецируя их на адресное пространство процесса.

DLL может импортировать функции и переменные из другой DLL. А значит, у нее может быть собственный раздел импорта, для которого необходимо повторить те же действия. В результате на инициализацию процесса может уйти довольно длительное время.

После отображения EXE-модуля и всех DLL-модулей на адресное пространство процесса его первичный поток готов к выполнению, и приложение начинает работу.

Недостатками неявной загрузки являются обязательная загрузка библиотеки, даже если обращение к ее функциям не будет происходить, и, соответственно, обязательное требование к наличию библиотеки при компоновке.

Явная загрузка DLL устраняет отмеченные выше недостатки за счет некоторого усложнения кода. Программисту приходится самому заботиться о загрузке DLL и подключении экспортируемых функций. Зато явная загрузка позволяет подгружать DLL по мере необходимости и дает возможность программе обрабатывать ситуации, возникающие при отсутствии DLL.

В случае явной загрузки процесс работы с DLL происходит в три этапа:

1. Загрузка DLL с помощью функции **LoadLibrary** (или ее расширенного аналога **LoadLibraryEx**). В случае успешной загрузки функция возвращает дескриптор hLib типа HMODULE , что позволяет в дальнейшем обращаться к этой DLL.
2. Вызовы функции **GetProcAddress** для получения указателей на требуемые функции или другие объекты. В качестве первого параметра функция **GetProcAddress** получает дескриптор hLib, в качестве второго параметра –адрес строки с именем импортируемой функции. Далее полученный указатель используется клиентом. Например, если это указатель на функцию, то осуществляется вызов нужной функции.
3. Когда загруженная динамическая библиотека больше не нужна, рекомендуется ее освободить, вызвав функцию **FreeLibrary.** Освобождение библиотеки не означает, что операционная система немедленно удалит ее из памяти. Задержка выгрузки предусмотрена на тот случай, когда эта же DLL через некоторое время вновь понадобится какому-то процессу. Но если возникнут проблемы с оперативной памятью, Windows в первую очередь удаляет из памяти освобожденные библиотеки.

Рассмотрим отложенную загрузку DLL. DLL отложенной загрузки (delay-load DLL) – это неявно связываемая DLL, которая не загружается до тех пор, пока код не обратится к какому-нибудь экспортируемому из нее идентификатору. Такие DLL могут быть полезны в следующих ситуациях:

- Если приложение использует несколько DLL, его инициализация может занимать длительное время, требуемое загрузчику для проецирования всех DLL на адресное пространство процесса. DLL отложенной загрузки позволяют решить эту проблему, распределяя загрузку DLL в ходе выполнения приложения.
- Если приложение предназначено для работы в различных версиях ОС, то часть функций может появиться лишь в поздних версиях ОС и не использоваться в текущей версии. Но если программа не вызывает конкретной функции, то DLL ей не нужна, и она может спокойно продолжать работу. При обращении же к несуществующей

функции можно предусмотреть выдачу пользователю соответствующего предупреждения.

Для реализации метода отложенной загрузки требуется дополнительно в список библиотек компоновщика добавить не только нужную библиотеку импорта MyLib.lib, но еще и системную библиотеку импорта delayimp.lib. Кроме этого, требуется добавить в опциях компоновщика флаг / delayload:MyLib.dll.

Перечисленные настройки заставляют компоновщик выполнить следующие операции:

- внедрить в EXE-модуль специальную функцию _delayLoadHelper;
- удалить MyLib.dll из раздела импорта исполняемого модуля, чтобы загрузчик операционной системы не пытался выполнить неявную загрузку этой библиотеки на этапе загрузки приложения;
- добавить в EXE-файл новый раздел отложенного импорта со списком функций, импортируемых из MyLib.dll;
- преобразовать вызовы функций из DLL к вызовам _delayLoadhelper.

На этапе выполнения приложения вызов функции из DLL реализуется обращением к _delayLoadHelper. Эта функция, используя информацию из раздела отложенного импорта, вызывает сначала LoadLibrary, а затем GetprocAddress. Получив адрес DLL-функции, _delayLoadHelper делает так, чтобы в дальнейшем эта DLL-функция вызывалась напрямую. Отметим, что каждая функция в DLL настраивается индивидуально при первом ее вызове.

4.3 Функция входа/выхода DLL

Предположим, что вашей библиотеке динамической компоновки требуется некоторая инициализация и деинициализация. Например, если в DLL при ее загрузке выделяются какие-то ресурсы, то при ее освобождении эти ресурсы также должны освобождаться.

Особое значение имеет деинициализация: поскольку при отключении DLL от адресного пространства процесса выделенная ею память сама собой не освобождается, а открытые файлы – не закрываются, DLL должна самостоятельно обеспечивать «уборку мусора».

Для решения указанных проблем вы можете включить в состав DLL специальную функцию точки входа DllMain. Эта функция вызывается операционной системой в следующих случаях:

- когда DLL проецируется на адресное пространство процесса (подключение DLL);

- когда процессом, загрузившим DLL, вызывается новый поток;

- когда завершается поток, принадлежащий процессу, который связан с DLL;

-когда процесс освобождает DLL (отключение DLL).

Функция точки входа DllMain имеет следующий прототип:

```
BOOL WINAPI DllMain(
      HINSTANCE hinstDLL,
// дескриптор DLL-модуля
DWORD fdwreason,
// флаг причины вызова функции
LPVOID lpvReserved
// дополнительная информация
) ;
```

В момент вызова функция получает информацию от операционной системы через свои параметры.

Первый параметр, hinstDLL, принимает значение дескриптора модуля DLL, являющееся, по сути, виртуальным адресом загрузки DLL. Если в библиотеке имеются вызовы функций, которым нужен данный дескриптор, необходимо сохранить значение hinstDLL в глобальной переменной.

Второй параметр, fdwReason, может принимать одно из следующих значений:

- DLL_PROCESS_ATTACH - уведомление о том, что DLL загружена в адресное пространство процесса либо в результате его старта, либо в результате вызова функции LoadLibrary;

- DLL_THREAD_ATTACH - уведомление о том, что текущий процесс создал новый поток. Это уведомление посылается всем DLL, подключенным к процессу. Вызов DllMain происходит в контексте нового потока;

- DLL_THREAD_DETACH - уведомление о том, что поток корректно завершается. Вызов DllMain происходит в контексте завершающегося потока;

DLL_PROCESS_DETACH - уведомление о том, что DLL отключается от адресного пространства процесса в результате одного из трех событий: а) неудачное завершение загрузки DLL; б) вызов функции FreeLibrary; в) завершение процесса.

Если при вызове функции используется первый параметр со значением DLL_PROCESS_ATTACH, то по значению третьего параметра можно выяснить, каким способом загружается DLL. При явной загруз-

ке параметр lvlReserved равен нулю, а при неявной загрузке принимает ненулевое значение.

Следует отметить следующие моменты работы с функцией входа-выхода DLL:

1. Поток, вызвавший DllMain со значением DLL_PROCESS_ATTACH, не вызывает повторно DllMain со значением DLL_THREAD_ATTACH.
2. Когда DLL загружается вызовом функции LoadLibrary, существующие потоки не вызывают DllMain для вновь загруженной библиотеки.
3. Функция DllMain не вызывается, если поток или процесс завершаются по причине вызова функции TerminateThread или TerminateProcess.

Поскольку функция DllMain должна обрабатывать все возможные причины своего вызова, то в ее коде обычно анализируется dwReason и выполняются необходимые действия по инициализации или освобождению ресурсов.

4.4 Примеры приложений, работающих с собственными DLL

Рассмотрим использование собственных DLL на нескольких примерах.

Пример 1. Рассмотрим три функции, включаемых в DLL: вычисление степени числа, преобразование числа в строку и вывод числа, хранящегося в регистре ***eax***.

Сначала напишем код, в котором реализуются функции DLL. Назовем файл MyDLL.asm.

```
.386
.model flat, stdcall
option casemap :none
include d:\masm32\include\windows.inc
include d:\masm32\include\user32.inc
include d:\masm32\include\kernel32.inc
includelib d:\masm32\lib\user32.lib
includelib d:\masm32\lib\kernel32.lib
szText MACRO Name, Text:VARARG
  LOCAL lbl
    jmp lbl
    Name db Text,0
    lbl:
  ENDM
```

```
m2m MACRO M1, M2
  push M2
  pop  M1
ENDM
return MACRO arg
  mov eax, arg
  ret
ENDM
.data
  hInst dd 0
  buf db 5 dup(0)
  szDisplayName db "Сообщение",0
.code
LibMain proc hInstDLL:DWORD, reason:DWORD,
            unused:DWORD
szText LmTitle,"tstdll's LibMain Function"
mov eax,hInstDLL
mov hInst,eax
  .if reason == DLL_PROCESS_ATTACH
       szText ATTACHPROCESS,"PROCESS_ATTACH"
       invoke MessageBox,NULL,ADDR ATTACHPROCESS,
              addr LmTitle,MB_OK
       return TRUE
  .elseif reason == DLL_PROCESS_DETACH
       szText DETACHPROCESS,"PROCESS_DETACH"
       invoke MessageBox,NULL,addr DETACHPROCESS,
              addr LmTitle,MB_OK
  .elseif reason == DLL_THREAD_ATTACH
       szText ATTACHTHREAD,"THREAD_ATTACH"
       invoke MessageBox,NULL,addr ATTACHTHREAD,
              addr LmTitle,MB_OK
  .elseif reason == DLL_THREAD_DETACH
       szText DETACHTHREAD,"THREAD_DETACH"
       invoke MessageBox,NULL,addr DETACHTHREAD,
              addr LmTitle,MB_OK
  .endif
ret
LibMain Endp
; Функции библиотеки
; степень a^b
step proc a :DWORD, b :DWORD
  mov eax,0
  mov ecx,b
  mov eax,1
l1: mul a
  loop l1
  return eax
```

```
step endp
; Перевод из числа в строку
IntToStr proc a :DWORD
  mov eax,a
  mov ebx,10
  mov ecx,0
  conv:
   .if eax & 8000h
     mov edx,0ffffh
   .else
     mov edx,0h
   .endif
  div ebx
  or dl,30h
  ror ecx,8
  mov cl,dl
  cmp eax,0
  jne conv
  ror ecx,8
  return ecx
IntToStr endp
; вывод содержимого регистра eax
out_eax proc
  mov buf[4],0
  mov buf[3],al
  shr eax,8
  mov buf[2],al
  shr eax,8
  mov buf[1],al
  shr eax,8
  mov buf[0],al
  invoke MessageBox,NULL,ADDR buf,
         ADDR szDisplayName,MB_OK
  return 0
out_eax endp
End LibMain
```

Экспортируемые функции из библиотеки описываются в файле MyDll.def:

```
LIBRARY MyDll
EXPORTS step
EXPORTS IntToStr
EXPORTS out_eax
```

Для демонстрации вызова функций из библиотеки напишем приложение, на форме которого расположим два элемента EditBox для ввода чисел и кнопку для выполнения вычислений и вывода результата с помощью функций из библиотеки. Текст приложения, хранимый в файле MyCallDll.asm, приведен ниже.

```
.386
.model flat, stdcall
option casemap :none
include d:\masm32\include\windows.inc
include d:\masm32\include\user32.inc
include d:\masm32\include\kernel32.inc
includelib d:\masm32\lib\user32.lib
includelib d:\masm32\lib\kernel32.lib
szText MACRO Name, Text:VARARG
  LOCAL lbl
   jmp lbl
   Name db Text,0
   lbl:
ENDM
m2m MACRO M1, M2
  push M2
  pop  M1
ENDM
return MACRO arg
  mov eax, arg
  ret
ENDM
WinMain PROTO :DWORD,:DWORD,:DWORD,:DWORD
WndProc PROTO :DWORD,:DWORD,:DWORD,:DWORD
TopXY PROTO   :DWORD,:DWORD
; функция для создания кнопки
PushButton PROTO :DWORD,:DWORD,:DWORD,:DWORD,:DWORD,
           :DWORD,:DWORD
; функция для создания EditBox
EditSl PROTO :DWORD,:DWORD,:DWORD,:DWORD,:DWORD,
           :DWORD,:DWORD
Ed1Proc PROTO :DWORD,:DWORD,:DWORD,:DWORD
.data
szDisplayName  db "Работа с DLL: посчитать a^b",0
CommandLine  dd 0
hWnd  dd 0
; дескриптор окна
hInstance  dd 0
; дескриптор приложения
aa  dd 0
```

```
step  dd 0
IntToStr  dd 0
out_eax  dd 0
hEdit1  dd 0
; дескриптор первого EditBox
hEdit2  dd 0
; дескриптор второго EditBox
nulbyte db 0
lpfnEd1Proc dd 0
; адрес оконной функции EditBox
num1  dd 0 ; первое число
num2  dd 0 ; второе число
.code
start:
  invoke GetModuleHandle, NULL
  mov hInstance, eax
  invoke GetCommandLine
  mov CommandLine, eax
  invoke WinMain,hInstance,NULL,CommandLine,
         SW_SHOWDEFAULT
  invoke ExitProcess,eax
WinMain proc hInst     :DWORD,
             hPrevInst :DWORD,
             CmdLine   :DWORD,
             CmdShow   :DWORD
  LOCAL wc   :WNDCLASSEX
  LOCAL msg  :MSG
  LOCAL Wwd  :DWORD
  LOCAL Wht  :DWORD
  LOCAL Wtx  :DWORD
  LOCAL Wty  :DWORD
  mov wc.cbSize, sizeof WNDCLASSEX
  mov wc.style, CS_HREDRAW or CS_VREDRAW \
                or CS_BYTEALIGNWINDOW
  mov wc.lpfnWndProc,  offset WndProc
  mov wc.cbClsExtra, NULL
  mov wc.cbWndExtra, NULL
  m2m wc.hInstance, hInst
  mov wc.hbrBackground,  COLOR_BTNFACE+1
  mov wc.lpszMenuName, NULL
  mov wc.lpszClassName,  offset szClassName
  invoke LoadIcon,hInst,500 ; идентификатор
;  пиктограммы из файла ресурсов
  mov wc.hIcon, eax
  invoke LoadCursor,NULL,IDC_ARROW
  mov wc.hCursor, eax
  mov wc.hIconSm, 0
```

```
  invoke RegisterClassEx, ADDR wc
; для вывода окна в центре экрана получим его размеры
  mov Wwd, 290
  mov Wht, 200
  invoke GetSystemMetrics,SM_CXSCREEN
  invoke TopXY,Wwd,eax
  mov Wtx, eax
  invoke GetSystemMetrics,SM_CYSCREEN
  invoke TopXY,Wht,eax
  mov Wty, eax
  szText szClassName,"CallDLL_Class"
  invoke CreateWindowEx,WS_EX_OVERLAPPEDWINDOW,
                        ADDR szClassName,
                        ADDR szDisplayName,
                        WS_OVERLAPPEDWINDOW,
                        Wtx,Wty,Wwd,Wht,
                        NULL,NULL,
                        hInst,NULL
  mov  hWnd,eax
  invoke ShowWindow,hWnd,SW_SHOWNORMAL
  invoke UpdateWindow,hWnd
StartLoop:
  invoke GetMessage,ADDR msg,NULL,0,0
  cmp eax, 0
  je ExitLoop
  invoke TranslateMessage, ADDR msg
  invoke DispatchMessage,  ADDR msg
  jmp StartLoop
ExitLoop:
  return msg.wParam
WinMain endp
; оконная функция
WndProc proc hWin:DWORD, uMsg   :DWORD,
             wParam :DWORD, lParam :DWORD
  LOCAL hLib  :DWORD
  .if uMsg == WM_COMMAND
;если нажата кнопка
        .if wParam == 500
            jmp @F
              libName  db "MyDll.dll",0
              FuncName db "step",0
              FuncName2 db "IntToStr",0
              sout_eax db "out_eax",0
            @@:
            invoke LoadLibrary,ADDR libName
            mov hLib, eax
            invoke GetProcAddress,hLib,ADDR FuncName2
```

```
            mov IntToStr,eax
            invoke GetProcAddress,hLib,ADDR sout_eax
            mov out_eax,eax
            invoke GetProcAddress,hLib,ADDR FuncName
            mov step,eax
            push dword ptr num1
            push dword ptr num2
            call step
            push eax
            call IntToStr
            call out_eax
            invoke FreeLibrary,hLib
        .endif
  .elseif uMsg == WM_CREATE
            szText Butn1,"Посчитать"
            invoke PushButton,ADDR Butn1,hWin,
                   20,20,140,25,500
            invoke EditS1,ADDR nulbyte,120,60,
                   100,23,hWin,200
            mov hEdit1, eax
            invoke EditS1,ADDR nulbyte,20,100,
                   100,23,hWin,200
            mov hEdit2, eax
            invoke SetWindowLong,hEdit1,GWL_WNDPROC,
                   Ed1Proc
            mov lpfnEd1Proc, eax
            invoke SetWindowLong,hEdit2,GWL_WNDPROC,
                   Ed1Proc
            mov lpfnEd1Proc, eax
  .elseif uMsg == WM_DESTROY
            invoke PostQuitMessage,NULL
           return 0
  .endif
  invoke DefWindowProc,hWin,uMsg,wParam,lParam
  ret
WndProc endp
; получение координат центра экрана
TopXY proc wDim:DWORD, sDim:DWORD
    shr sDim, 1
    shr wDim, 1
    mov eax, wDim
    sub sDim, eax
    return sDim
TopXY endp
; создание кнопки
PushButton proc lpText:DWORD,hParent:DWORD,
           a:DWORD,b:DWORD,wd:DWORD,ht:DWORD,ID:DWORD
```

```
  szText btnClass,"BUTTON"
  invoke CreateWindowEx,0, ADDR btnClass,lpText,
            WS_CHILD or WS_VISIBLE,
            a,b,wd,ht,hParent,ID,hInstance,NULL
  ret
PushButton endp
; создание EditBox
EditSl proc szMsg:DWORD,a:DWORD,b:DWORD,
            wd:DWORD,ht:DWORD,hParent:DWORD,ID:DWORD
  szText slEdit,"EDIT"
  invoke CreateWindowEx,WS_EX_CLIENTEDGE,
         ADDR slEdit,szMsg,
         WS_VISIBLE or WS_CHILDWINDOW or \
         ES_AUTOHSCROLL or ES_NOHIDESEL,
         a,b,wd,ht,hParent,ID,hInstance,NULL
  ret
EditSl endp
; оконная функция для EditBox
Ed1Proc proc hCtl   :DWORD,
             uMsg   :DWORD,
             wParam :DWORD, lParam :DWORD
; формирование числа из строки
  .if uMsg == WM_CHAR
    .if wParam == 8              ; нажат backspace
        mov eax,hEdit1
         .if hCtl == eax
             mov num1,0
         .else
             mov num2,0
         .endif
            jmp accept
    .endif
    .if wParam < "0"
        return 0
    .endif
    .if wParam > "9"
        return 0
    .endif
     mov eax,hEdit1
     .if hCtl == eax
          .if num1 == 0
              mov ebx,wParam
              and ebx,0fh
              mov num1,ebx
          .else
              mov eax,num1
              mov bx,10
```

```
            mul bx
            mov ebx,wParam
            and bx,0fh
            add ax,bx
            mov num1,eax
         .endif
    .elseif
          .if num2 == 0
            mov ebx,wParam
            and ebx,0fh
            mov num2,ebx
          .else
             mov eax,num2
             mov bx,10
             mul bx
             mov ebx,wParam
             and bx,0fh
             add ax,bx
             mov num2,eax
          .endif
    .endif
  .endif
  accept:
    invoke CallWindowProc,lpfnEd1Proc,hCtl,uMsg,
          wParam,lParam
  ret
Ed1Proc endp
end start
```

Создание библиотеки MyDll.dll осуществляется при запуске bat-файла вида:

```
@echo off
if exist MyDll.obj del MyDll.obj
if exist MyDll.dll del MyDll.dll
d:\masm32\bin\ml /c /coff MyDll.asm
d:\masm32\bin\Link /SUBSYSTEM:WINDOWS /DLL
/DEF:MyDll.def MyDll.obj
pause
```

Создание исполняемого файла осуществляется при запуске bat-файла:

```
@echo off
if exist MyCallDll.obj del MyCallDll.obj
if exist MyCallDll.exe del MyCallDll.exe
if exist rsrc.res del rsrc.res
```

```
if exist rsrc.obj del rsrc.obj
d:\masm32\BIN\Rc.exe /v rsrc.rc
d:\masm32\BIN\Cvtres.exe /machine:ix86 rsrc.res
d:\masm32\bin\ml /c /coff MyCallDll.asm
d:\masm32\bin\link.exe /SUBSYSTEM:WINDOWS MyCallDll.obj
rsrc.obj
pause
```

Пример работы приложения приведен на рисунке 3.5.

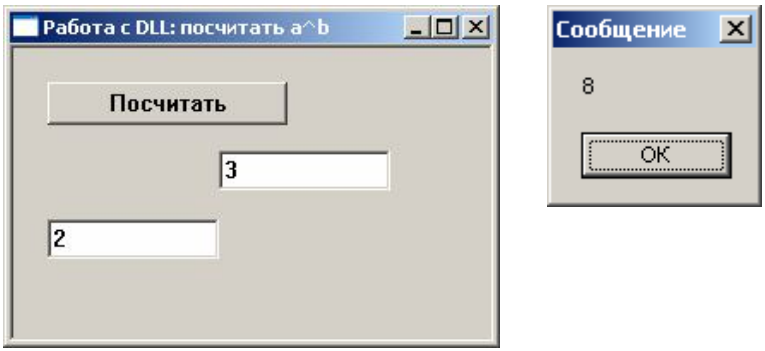


Рисунок 3.5

Пример 2. Пусть даны три числа. Напишем функции для расчета среднего арифметического, произведения, суммы квадратов чисел, а также для проверки, являются ли 2 и 3 числа делителями первого. Все действия с числами оформим в виде меню, которое опишем в файле ресурсов:

```
500 ICON MOVEABLE PURE LOADONCALL DISCARDABLE
        "MAINICON.ICO"
600 MENUEX MOVEABLE IMPURE LOADONCALL DISCARDABLE
BEGIN
    POPUP "&Функции",,,0
    BEGIN
        MENUITEM "&Делители",2000
        MENUITEM "&Среднее арифм",2100
        MENUITEM "&Умножение",2200
        MENUITEM "&Сумма квадратов",2300
        MENUITEM "&Очистить",2400
        MENUITEM "&Выход", 1000
    END
END
```

Исходные числа и результаты расчетов будем помещать в строки редактирования элементов EditBox, поэтому в качестве параметров функции DLL будут получать дескрипторы этих элементов. Назовем нашу библиотеку tstdll. Ее описание в файле tstdll.def имеет вид:

```
LIBRARY tstdll
EXPORTS Deliteli
EXPORTS SrArifm
EXPORTS FindMult
EXPORTS FindMultSqr
```

Текст, содержащий описание функций библиотеки, приведен ниже.

```
.386
.model flat, stdcall
option casemap :none
include d:\masm32\include\windows.inc
include d:\masm32\include\user32.inc
include d:\masm32\include\kernel32.inc
include d:\masm32\include\masm32.inc
includelib d:\masm32\lib\masm32.lib
includelib d:\masm32\lib\user32.lib
includelib d:\masm32\lib\kernel32.lib
szText MACRO Name, Text:VARARG
  LOCAL lbl
    jmp lbl
      Name db Text,0
    lbl:
ENDM
m2m MACRO M1, M2
  push M2
  pop  M1
ENDM
return MACRO arg
  mov eax, arg
  ret
ENDM
.data
  buff db 50 dup(0)
; буфер для преобразования числа в строку
.code
LibMain proc hInstDLL:DWORD, reason:DWORD,
            unused:DWORD
szText LmTitle,"tstdll's LibMain Function"
  .if reason == DLL_PROCESS_ATTACH
           return TRUE
```

```
        .endif
        ret
LibMain Endp
; функция, получающая три числа из строк
; редактирования элементов EditBox с дескрипторами
; hE1, hE2, hE3, проверяющая являются ли второе и
; третье числа делителями первого и выдающая сообщение
; о результатах проверки
Deliteli proc hE1:DWORD,hE2:DWORD,hE3:DWORD
  LOCAl chislo1:DWORD
; число 1
  LOCAl chislo2:DWORD
; число 2
  LOCAl chislo3:DWORD
; число 3
  LOCAl flag1:DWORD
  LOCAl flag2:DWORD
; вспомогательные флаги для запоминания результатов
; проверки на делимость
  invoke SendMessage, hE1,WM_GETTEXT,50,addr buff
; забрать строку из EditBox для первого числа
; и поместить в buff
  invoke atodw,addr buff
; преобразовать строку из buff в число, записываемое в
; регистр eax
  mov chislo1,eax
; запомнить первое число в chislo1
; аналогичные действия со вторым и третьим числом
  invoke SendMessage, hE2,WM_GETTEXT,50,addr buff
  invoke atodw,addr buff
  mov chislo2,eax
  invoke SendMessage, hE3,WM_GETTEXT,50,addr buff
  invoke atodw,addr buff
  mov chislo3,eax
  .if chislo1 == 0
    szText Msg,
     "Число равно 0!У него нет ненулевых делителей!"
    szText Ttl1,"Решение"
    invoke MessageBox,NULL,addr Msg,addr Ttl1,MB_OK
  .else
    .if chislo2 == 0
       szText Msg2,"Второе число равно 0!"
       szText Ttl2,"Решение"
       invoke MessageBox,NULL,addr Msg2,addr Ttl2,
              MB_OK
    .else
       mov flag1,0
```

```
mov eax,chislo1
mov ebx,chislo2
cdq
div ebx
.if edx == 0
   mov flag1,1
.endif
mov flag2,0
.if chislo3 == 0
   szText Msg3,"Второе число равно 0!"
   szText Ttl3,"Решение"
   invoke MessageBox,NULL,addr Msg3,
          addr Ttl3,MB_OK
.else
   mov eax,chislo1
   mov ebx,chislo3
   cdq
   div ebx
   .if edx == 0
      mov flag2,1
   .endif
.endif
.if flag1 == 1
  .if flag2 == 1
    szText Msg4,"Оба числа явл делителями!"
    szText Ttl4,"Решение"
    invoke MessageBox,NULL,addr Msg4,
           addr Ttl4,MB_OK
  .else
    szText Msg5,"Только второе число делитель!"
    szText Ttl5,"Решение"
    invoke MessageBox,NULL,addr Msg5,
           addr Ttl5,MB_OK
  .endif
.else
  .if flag2 == 1
    szText Msg6,"Только третье число делитель!"
    szText Ttl6,"Решение"
    invoke MessageBox,NULL,addr Msg6,
           addr Ttl6,MB_OK
  .else
    szText Msg7,"Числа не явл делителями!"
    szText Ttl7,"Решение"
    invoke MessageBox,NULL,addr Msg7,
           addr Ttl7,MB_OK
  .endif
.endif
```

```
    .endif
  .endif
  ret
Deliteli endp
; функция, находящая среднее арифметическое трех чисел
; из строк редактирования трех элементов EditBox и
; выводящая результат в четвертый EditBox
SrArifm proc hE1:DWORD,hE2:DWORD,hE3:DWORD,hE4:DWORD
  LOCAl chislo1:DWORD
  LOCAl chislo2:DWORD
  LOCAl chislo3:DWORD
  LOCAl rezult:DWORD
  invoke SendMessage, hE1,WM_GETTEXT,50,addr buff
  invoke atodw,addr buff
  mov chislo1,eax
  invoke SendMessage, hE2,WM_GETTEXT,50,addr buff
  invoke atodw,addr buff
  mov chislo2,eax
  invoke SendMessage, hE3,WM_GETTEXT,50,addr buff
  invoke atodw,addr buff
  mov chislo3,eax
  mov eax,chislo1
  mov edx,chislo2
  add eax,edx
  mov edx,chislo3
  add eax,edx
  mov ebx,3
  cdq
  idiv ebx
  mov rezult,eax
  invoke dwtoa,rezult,addr buff
  invoke  SendMessage,hE4,WM_SETTEXT,0,addr buff
; запись строки из buff в строку редактирования
; однострочного редактора с дескриптором hE4
  ret
SrArifm endp
; функция для поиска произведения трех чисел
; из строк редактирования трех элементов EditBox и
; выводящая результат в четвертый EditBox
FindMult proc hE1:DWORD,hE2:DWORD,hE3:DWORD,hE4:DWORD
  LOCAl chislo1:DWORD
  LOCAl chislo2:DWORD
  LOCAl chislo3:DWORD
  LOCAl rezult:DWORD
  invoke SendMessage, hE1,WM_GETTEXT,50,addr buff
  invoke atodw,addr buff
  mov chislo1,eax
```

```
  invoke SendMessage, hE2,WM_GETTEXT,50,addr buff
  invoke atodw,addr buff
  mov chislo2,eax
  invoke SendMessage, hE3,WM_GETTEXT,50,addr buff
  invoke atodw,addr buff
  mov chislo3,eax
  mov eax,chislo1
  mov ebx,chislo2
  mov ecx,chislo3
  imul ebx
  imul ecx
  mov rezult,eax
  invoke dwtoa,rezult,addr buff
  invoke  SendMessage,hE4,WM_SETTEXT,0,addr buff
  ret
FindMult endp
; функция для нахождения суммы квадратов трех чисел
; из строк редактирования трех элементов EditBox и
; выводящая результат в четвертый EditBox
FindMultSqr proc hE1:DWORD,hE2:DWORD,hE3:DWORD,
                 hE4:DWORD
  LOCAl chislo1:DWORD
  LOCAl chislo2:DWORD
  LOCAl chislo3:DWORD
  LOCAl rezult:DWORD
  LOCAl kv1:DWORD
  LOCAl kv2:DWORD
  LOCAl kv3:DWORD
  invoke SendMessage, hE1,WM_GETTEXT,50,addr buff
  invoke atodw,addr buff
  mov chislo1,eax
  invoke SendMessage, hE2,WM_GETTEXT,50,addr buff
  invoke atodw,addr buff
  mov chislo2,eax
  invoke SendMessage, hE3,WM_GETTEXT,50,addr buff
  invoke atodw,addr buff
  mov chislo3,eax
  mov eax,chislo1
  imul eax
  mov kv1,eax
  mov eax,chislo2
  imul eax
  mov kv2,eax
  mov eax,chislo3
  imul eax
  mov kv3,eax
  mov eax,kv1
```

```
  mov ebx,kv2
  add eax,ebx
  mov ebx,kv3
  add eax,ebx
  mov rezult,eax
  invoke dwtoa,rezult,addr buff
  invoke  SendMessage,hE4,WM_SETTEXT,0,addr buff
  ret
FindMultSqr endp
End LibMain
```

Текст, содержащий вызов функций библиотеки, приведен ниже.

```
.386
.model flat, stdcall
option casemap :none
include d:\masm32\include\windows.inc
include d:\masm32\include\user32.inc
include d:\masm32\include\masm32.inc
include d:\masm32\include\kernel32.inc
includelib d:\masm32\lib\masm32.lib
includelib d:\masm32\lib\user32.lib
includelib d:\masm32\lib\kernel32.lib
includelib tstdll.lib
szText MACRO Name, Text:VARARG
   LOCAL lbl
     jmp lbl
       Name db Text,0
     lbl:
ENDM
m2m MACRO M1, M2
   push M2
   pop  M1
ENDM
return MACRO arg
   mov eax, arg
   ret
ENDM
WinMain PROTO :DWORD,:DWORD,:DWORD,:DWORD
WndProc PROTO :DWORD,:DWORD,:DWORD,:DWORD
EditSl PROTO  :DWORD,:DWORD,:DWORD,:DWORD,:DWORD,
              :DWORD,:DWORD
Deliteli PROTO :DWORD,:DWORD,:DWORD
SrArifm PROTO :DWORD,:DWORD,:DWORD,:DWORD
FindMult PROTO :DWORD,:DWORD,:DWORD,:DWORD
FindMultSqr PROTO :DWORD,:DWORD,:DWORD,:DWORD
.data
```

```
  szDisplayName db "DLL-функции",0
  CommandLine   dd 0
  hWnd  dd 0
  hInstance  dd 0
  nulbyte db 0
  hEdit1  DWORD 0
  hEdit2  DWORD 0
  hEdit3  DWORD 0
  hEdit4  DWORD 0
  hEdit5  DWORD 0
  hEdit6  DWORD 0
  buff db 50 dup(0)
.code
start:
  invoke GetModuleHandle, NULL
  mov hInstance, eax
  invoke GetCommandLine
  mov CommandLine, eax
  invoke WinMain,hInstance,NULL,CommandLine,
         SW_SHOWDEFAULT
  invoke ExitProcess,eax
WinMain proc hInst     :DWORD,
             hPrevInst :DWORD,
             CmdLine   :DWORD,
             CmdShow   :DWORD
  LOCAL wc:WNDCLASSEX
  LOCAL msg:MSG
  LOCAL Wwd:DWORD
  LOCAL Wht:DWORD
  LOCAL Wtx:DWORD
  LOCAL Wty:DWORD
  mov wc.cbSize, sizeof WNDCLASSEX
  mov wc.style, CS_HREDRAW or CS_VREDRAW \
                or CS_BYTEALIGNWINDOW
  mov wc.lpfnWndProc, offset WndProc
  mov wc.cbClsExtra, NULL
  mov wc.cbWndExtra, NULL
  m2m wc.hInstance, hInst
  mov wc.hbrBackground, COLOR_BTNFACE+1
  mov wc.lpszMenuName, NULL
  mov wc.lpszClassName, offset szClassName
  invoke LoadIcon,hInst,500
  mov wc.hIcon, eax
  invoke LoadCursor,NULL,IDC_ARROW
  mov wc.hCursor, eax
  mov wc.hIconSm, 0
  invoke RegisterClassEx, ADDR wc
```

```
  szText szClassName,"MyClass"
  invoke CreateWindowEx,WS_EX_OVERLAPPEDWINDOW,
         ADDR szClassName, ADDR szDisplayName,
         WS_OVERLAPPEDWINDOW, 0,0,480,480,
         NULL,NULL, hInst,NULL
  mov hWnd,eax
  invoke LoadMenu,hInst,600
  invoke SetMenu,hWnd,eax
  invoke ShowWindow,hWnd,SW_SHOWNORMAL
  invoke UpdateWindow,hWnd
  StartLoop:
   invoke GetMessage,ADDR msg,NULL,0,0
   cmp eax, 0
   je ExitLoop
   invoke TranslateMessage, ADDR msg
   invoke DispatchMessage,  ADDR msg
   jmp StartLoop
  ExitLoop:
  return msg.wParam
WinMain endp
WndProc proc hWin   :DWORD,
             uMsg   :DWORD,
             wParam :DWORD,
             lParam :DWORD
  LOCAL hLib  :DWORD
  LOCAL nol:DWORD
  .if uMsg == WM_COMMAND
      .if wParam == 1000
         invoke SendMessage,hWin,WM_SYSCOMMAND,
                SC_CLOSE, NULL
      .elseif wParam == 2000
         invoke Deliteli,hEdit1,hEdit2,hEdit3
      .elseif wParam == 2100
         invoke SrArifm,hEdit1,hEdit2,hEdit3,hEdit4
; результат вычисления среднего арифметического – в
; четвертый EditBox
      .elseif wParam == 2200
         invoke FindMult,hEdit1,hEdit2,hEdit3,hEdit5
; результат вычисления произведения трех чисел –
; в пятый EditBox
      .elseif wParam == 2300
        invoke FindMultSqr,hEdit1,hEdit2,hEdit3,hEdit6
; результат вычисления суммы квадратов трех чисел –
; в шестой EditBox
      .elseif wParam == 2400
; очистка всех элементов EditBox
         mov nol,0
```

```
          invoke dwtoa,nol,addr buff
          invoke SendMessage, hEdit1,WM_SETTEXT,0,
                 addr buff
          invoke SendMessage, hEdit2,WM_SETTEXT,0,
                 addr buff
          invoke SendMessage, hEdit3,WM_SETTEXT,0,
                 addr buff
          invoke SendMessage, hEdit4,WM_SETTEXT,0,
                 addr buff
          invoke SendMessage, hEdit5,WM_SETTEXT,0,
                 addr buff
          invoke SendMessage, hEdit6,WM_SETTEXT,0,
                 addr buff
      .endif
  .elseif uMsg == WM_CREATE
      invoke EditSl,ADDR nulbyte,100,10,50,23,hWin,200
      mov hEdit1, eax
      invoke EditSl,ADDR nulbyte,100,50,50,23,hWin,200
      mov hEdit2, eax
      invoke EditSl,ADDR nulbyte,100,90,50,23,hWin,200
      mov hEdit3, eax
      invoke EditSl,ADDR nulbyte,160,10,50,23,hWin,200
      mov hEdit4, eax
      invoke EditSl,ADDR nulbyte,160,50,50,23,hWin,200
      mov hEdit5, eax
      invoke EditSl,ADDR nulbyte,160,90,50,23,hWin,200
      mov hEdit6, eax
  .elseif uMsg == WM_DESTROY
      invoke PostQuitMessage,NULL
      return 0
  .endif
  invoke DefWindowProc,hWin,uMsg,wParam,lParam
  ret
WndProc endp
; Создание EditBox
EditSl proc szMsg:DWORD,a:DWORD,b:DWORD,
            wd:DWORD,ht:DWORD,hParent:DWORD,ID:DWORD
  szText slEdit,"EDIT"
  invoke CreateWindowEx,WS_EX_CLIENTEDGE,
         ADDR slEdit,szMsg,
         WS_VISIBLE or WS_CHILDWINDOW or \
         ES_AUTOHSCROLL or ES_NOHIDESEL,
         a,b,wd,ht,hParent,ID,hInstance,NULL
  ret
EditSl endp
end start
```

4.5 Контрольные вопросы

1. Что такое DLL?
2. Зачем нужен раздел экспорта?
3. Какие преимущества использования DLL?
4. Как защищаются данные DLL от ошибок приложений?
5. Какие способы загрузки DLL Вы знаете?
6. Как осуществляется неявная загрузка DLL?
7. Как выполняется загрузка exe-файла, использующего DLL при неявном связывании?
8. Каковы недостатки неявной загрузки?
9. Чем отличается явная загрузка от неявной?
10. Каково назначение функции **LoadLibrary**?
11. Зачем используется функция **GetProcAdress**?
12. Перечислите этапы работы с DLL при явной загрузке.
13. Что такое отложенная загрузка DLL?
14. Зачем используется отложенная загрузка?
15. Как организуется отложенная загрузка?
16. Зачем нужна функция входа-выхода DLL?
17. В каких случаях вызывается DLLMain?
18. Какие параметры у функции входа-выхода?
19. Какие причины вызова DLLMain Вы знаете?
20. Как узнать, каким способом загружается DLL?

Список использованных источников:

1. Вильямс М. Программирование в Windows 2000. Энциклопедия пользователя. – Киев: ДиаСофт, 2000. – 640 с.
2. Ганеев Р.М. Проектирование интерфейса пользователя средствами Win32 API: [учеб. пособие для вузов].-М.: Горячая линия - Телеком, 2007.-357 с.: ил.
3. Джонсон М. Харт. Системное программирование в среде Win-32, 2-е изд.: Пер. с англ.: - М.: Издательский дом «Вильямс», 2001. – 464 с.: ил. – Парал. тит. англ.
4. Пирогов В.Ю. Ассемблер для Windows. – 2-е изд., перераб. и доп. – СПб.: БХВ – Петербург, 2003. – 656 с.: ил.
5. Побегайло А.П. Системное программирование в WINDOWS. – Изд-во БХВ-Петербург, 2006. – 1056 с.
6. Рихтер Дж. Windows для профессионалов: создание эффективных Win-32 приложений с учетом специфики 64-разрядной версии Windows. – СПб: Питер, 2001. – 752 с.
7. Румянцев П.В. Азбука программирования в Win-32 API. – М.: Горячая линия – Телеком, 2001. – 312 с.
8. Саймон Р. Microsoft Windows 2000 API. Энциклопедия программиста. – Киев: ДиаСофт, 2001. – 1008 с.
9. Финогенов К.Г. Основы программирования. – М.: ДИАЛОГ-МИФИ, 2002. – 146 с.
10. Фленов М.Е. Программирование на C++ глазами хакера. – СПб: БХВ-Петербург, 2007. – 336 с.
11. Щупак Ю.А. Win32 API. Разработка приложений для Windows. - СПб, ПИТЕР. 1-е издание, 2008 - 592 с.

Лариса Иннокентьевна Сучкова

Win32 API: основы программирования

Издано в авторской редакции

Подписано в печать 21.09.2010. Формат 60x84 1/16.
Печать – цифровая. Усл.п.л. 9,53.
Тираж 70 экз. Заказ 2010 - 437

Отпечатано в типографии АлтГТУ,
656038, г. Барнаул, пр-т Ленина, 46
тел.: (8–3852) 36–84–61

Лицензия на полиграфическую деятельность
ПЛД №28–35 от 15.07.97 г.