

Министерство образования Республики Беларусь  
БЕЛОРУССКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ  
УНИВЕРСИТЕТ

---

Кафедра «Программное обеспечение вычислительной техники  
и автоматизированных систем»

ОПЕРАЦИОННЫЕ СИСТЕМЫ И СИСТЕМНОЕ  
ПРОГРАММИРОВАНИЕ

Методические указания к лабораторным работам  
для студентов специальностей 1-40 01 01  
«Программное обеспечение информационных технологий»  
и 1-40 01 02 «Информационные системы и технологии»

М и н с к  
Б Н Т У  
2 0 1 1

УДК 004.434 (075.8)

ББК 32.973 я7

О 60

Составитель Н.А. Разорёнов

Рецензенты:

В.А. Бородуля, О.В. Бугай

- О 60 Операционные системы и системное программирование: методические указания к лабораторным работам для студентов специальностей 1-40 01 01 «Программное обеспечение информационных технологий» и 1-40 01 02 «Информационные системы и технологии» / сост.: Н.А. Разорёнов. – Минск: БНТУ, 2011. – 94 с.

ISBN 978-985-525-539-1.

Приведен теоретический материал по выполнению лабораторных работ по дисциплинам «Операционные системы и системное программирование» и «Системное программирование». Рассматриваются вопросы организации и принципы программирования в операционных системах (ОС) семейства Windows, графический оконный интерфейс, использование аппаратных и программных средств современных ОС, предназначенных для поддержки многозадачных операционных систем, архитектура файловых подсистем FAT и NTFS и управления файлами, создание, управление и взаимодействие процессов.

Указания могут быть полезны студентам специальностей, связанных с программированием, и лицам, которые занимаются разработкой программного обеспечения на базе прикладного программного интерфейса операционных систем.

УДК 004.434 (075.8)

ББК 32.973 я7

ISBN 978-985-525-539-1

© БНТУ, 2011

## СОДЕРЖАНИЕ

Основные требования к оформлению и содержанию отчета о лабораторной работе. ....	4
Лабораторная работа № 1. Простейшее приложение на базе WIN32/64 API. ....	7
Лабораторная работа № 2. Диалоговые окна. ....	19
Лабораторная работа № 3. Программирование внешних устройств. ....	36
Лабораторная работа № 4. Работа с файлами. ....	51
Лабораторная работа № 5. Файловая система FAT. ....	64
Лабораторная работа № 6. Файловая система NTFS. ....	73
Лабораторная работа № 7. Процессы (часть 1). ....	82
Лабораторная работа № 8. Процессы (часть 2). ....	88
Литература. ....	93

## **Основные требования к оформлению и содержанию отчета о лабораторной работе**

При оформлении отчета о работе следует соблюдать следующие требования:

1. Шрифт – Times New Roman, 12–14 пт, через полтора интервала. Параметры страницы: формат А4, левое поле 30 мм, правое поле 10 мм, верхнее и нижнее поля 20 мм. Абзацы 15–17 мм, одинаковые по всему тексту. Страницы следует нумеровать в верхнем правом углу. Номер страницы на титульном листе не ставится, но включается в общую нумерацию страниц.

2. Отчёт оформляется персонально и самостоятельно, представляется к защите в установленный срок в бумажном/электронном виде перед защитой лабораторной работы. Форма отчета устанавливается преподавателем. Выполненные лабораторные работы и отчеты сохраняются до конца семестра. Объем отчета 5–7 листов формата А4. Отчет может содержать приложения.

3. Отчет должен содержать следующие листы и пункты:

1-й лист – титульный лист (пример оформления титульного листа отчета приведен на рис. 1);

2-й и последующие листы отчета содержат пункты:

1) Цель работы.

2) Изучаемые вопросы.

3) Постановка задачи.

} берутся из описания работы

4) Ход выполнения работы (содержит подпункты, комментирующие фрагменты кода разработанной программы, которые раскрывают изучаемый вопрос. В подпунктах допускается приводить краткие теоретические сведения, схемы, рисунки, фрагменты дампов изучаемых объектов и т. д.).

Вариант примера ответа на 9-й вопрос хода выполнения 5-й лабораторной работы приведен на рис. 2.

5) Результаты работы программного обеспечения.

6) Выводы (не менее шести пунктов).

7) Приложения (при необходимости).

БЕЛОРУССКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ  
УНИВЕРСИТЕТ

Факультет информационных технологий  
и робототехники

Кафедра «Программное обеспечение вычислительной  
техники и автоматизированных систем»

О Т Ч Е Т

о лабораторной работе № N

Дисциплина  
«Системное программирование»

Тема  
«Файловая система FAT»

Выполнил: студент гр.107215

Трошкин Р.Ю.

Проверил:

Разорёнов Н.А.

Минск 20 х

Рис. 1. Пример оформления титульного листа отчета

4.9 Программно определить дату и время создания файла с именем NAME, подчиненного корневому каталогу ROOT.

Объявляем две структуры с битовыми полями для атрибутов время и дата:

```
struct sModifyTime
{
    unsigned short timeSeconds:5;
    unsigned short timeMinutes:6;
    unsigned short timeHours:5;
};
struct sModifyDate
{
    unsigned short day:5;
    unsigned short month:4;
    unsigned short year:7;
};
```

Считываем с файла дату и время, описание структуры sIndex приведено в отчете выше:

```
ReadFile(hFile, &sIndex.time, 2, &length, NULL);
ReadFile(hFile, &sIndex.date, 2, &length, NULL);
Форматируем строку для вывода даты и времени:
j += sprintf(buffer+j, "Дата создания\модификации
файла : %d.%d.%d\n",
sIndex.date.day, sIndex.date.month, sIndex.date.year
+ 1980);
j += sprintf(buffer+j, "Время создания\модификации
файла : %d.%d.%d\n",
sIndex.time.timeHours, sIndex.time.timeMinutes,
sIndex.time.timeSeconds);
```

Рис. 2. Пример ответа

## *Лабораторная работа № 1*

### **ПРОСТЕЙШЕЕ ПРИЛОЖЕНИЕ НА БАЗЕ WIN32/64 API**

**Цель работы:** изучить основы создания и управления окнами Windows-приложений на базе WIN32/64 API.

#### **Изучаемые вопросы**

1. Архитектура WIN32/64 программы.
2. Цикл обработки сообщений (GetMessage).
3. Оконная функция, назначение, параметры, обработка сообщений.
4. Структура оконного класса, поля, их назначение, инициализация.
5. Классы окон, регистрация оконного класса.
6. Создание экземпляра оконного класса.
7. Отображение окна на экране.
8. Управление положением и размерами окна.
9. Обновление оконной области на экране.
10. Установки фона.
11. Курсор приложения. Типы курсоров, их прикрепление к приложению.

#### ***Постановка задачи***

Разработать приложение, в котором регистрируются оконные классы, создаются окна классов в соответствии с вариантом задания.

#### **Теоретические сведения**

##### ***Простейшее Windows-приложение на базе Win32 API***

Ниже приведена минимальная программа-каркас Windows приложения с использованием обработчиков основных событий.

```
#include <windows.h>
#include <string.h>
#include <stdio.h>
```

```

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

char szWinName[] = "MyWin";          // Имя класса окна
char str[255] = "";                  // Строка вывода
int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPre-
vInst, LPSTR lpszArgs, int nWinMode)
{
    HWND hwnd;                       // Дескриптор окна
    MSG msg;                          // Структура сообщения Windows
    WNDCLASSEX wcl;                   // Структура, определяющая класс окна
    /* Определение класса окна */
    wcl.hInstance = hThisInst; // Работа с данным экземпляром
    wcl.lpszClassName = szWinName; // Имя класса окна
    wcl.lpfnWndProc = WindowFunc; // Оконная функция
    wcl.style = 0; // Использовать стиль окна по умолчанию
    wcl.cbSize = sizeof(WNDCLASSEX); // Установка размера
    // WNDCLASSEX
    wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION); // Большие иконки
    wcl.hIconSm = LoadIcon(NULL, IDI_WINLOGO); // Маленькие
    // иконки
    wcl.hCursor = LoadCursor(NULL, IDC_ARROW); // Стиль курсора
    wcl.lpszMenuName = NULL; // Меню нет
    wcl.cbClsExtra = 0; // Дополнительной информации нет
    wcl.cbWndExtra = 0; // Дополнительной информации нет
    wcl.hbrBackground = (HBRUSH)GetStockObject(HOLLOW_BRUSH);
    // Установка фона окна
    /* Регистрация класса окна */
    if (!RegisterClassEx(&wcl)) return 0;
    /* Создаём само окно */
    hwnd = CreateWindow(szWinName, // Имя класса окна
    "API application", // Заголовок
    WS_OVERLAPPEDWINDOW // Стиль окна - нормальный
    CW_USEDEFAULT, // Координата X - выбирает Windows
    CW_USEDEFAULT, // Ширина - выбирает Windows
    CW_USEDEFAULT, // Высота - выбирает Windows
    HWND_DESKTOP, // Нет родительского окна
    NULL, // Нет меню
    hThisInst, // Работа с данным экземпляром программы
    NULL // Дополнительные аргументы отсутствуют
    );
    /* Вывод окна */
    ShowWindow(hwnd, nWinMode);
    UpdateWindow(hwnd);

    /* Создание цикла обработки сообщений */
    while (GetMessage(&msg, NULL, 0, 0))
    {

```



```

TranslateMessage(&msg); //Разрешить использование клавиатуры
DispatchMessage(&msg); // Вернуть управление Windows
}
return msg.wParam;
} // WinMain

```

/\* Оконная функция обработки сообщений, полученных из очереди сообщений. \*/

```

LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message,
WPARAM wParam, LPARAM lParam)
{
    HDC hdc;
    switch (message)
    {
        case WM_CHAR: Код обработчика // Символьное сообщение
                        //от клавиатуры

        break;
        case WM_LBUTTONDOWN: Код обработчика // Обработчик
                        //нажатия левой кнопки мыши

        break;
        case WM_RBUTTONDOWN: Код обработчика // Обработчик
                        //нажатия правой кнопки мыши

        break;
        case WM_DESTROY: // Завершение программы
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc(hwnd, message, wParam,
lParam);
    }
    return 0;
}

```

## ***Структура оконного класса: элементы и их назначение***

Структура WNDCLASSEX содержит атрибуты класса окна, который регистрируется с помощью функции *RegisterClass*.

### **Структура WNDCLASSEX:**

```
typedef struct _WNDCLASSEX {
    UINT style;           //Определяет стиль окна
    WNDPROC lpfnWndProc;  //Указатель на процедуру окна
    int cbClsExtra;       //Определяет количество
                        //дополнительных байт для размеще-
                        //ния структуры оконного класса
    int cbWndExtra;       //Определяет количество дополни-
                        //тельных байт для размещения
                        //экземпляра окна
    HANDLE hInstance;     //Указатель на экземпляр окна
    HICON hIcon;          //Дескриптор иконки
    HCURSOR hCursor;      //Дескриптор курсора
    HBRUSH hbrBackground; //Дескриптор кисти фона окна
    LPCTSTR lpszMenuName; //Указатель на нуль
                        //терминированную строку, определяющую
                        // имя ресурса для класса меню
    LPCTSTR lpszClassName; //Строка, определяющая
                        // имя класса окна
    HICON hIconSm;        //Дескриптор маленькой иконки,
                        //ассоциированной с классом окна
} WNDCLASSEX;
```

## ***Регистрация в Windows оконного класса***

Регистрация оконного класса производится с помощью функции *RegisterClassEx* для последующего использования посредством вызова функции *CreateWindowEx*.

```
ATOM RegisterClassEx(
    CONST WNDCLASSEX *lpwscx //Адрес структуры, содержащей
                            //информацию о классе
);
```

Если функция завершена успешно, то она возвращает значение атома, который однозначно определяет регистрируемый класс. В случае неуспеха функция возвращает ноль.

## Создание экземпляров оконных классов

Функция *CreateWindow* создает перекрывающиеся, всплывающие или дочерние окна.

```
HWND CreateWindow(  
    LPCTSTR lpClassName,        // указатель на  
                                // зарегистрированный класс окна  
    LPCTSTR lpWindowName,      // указатель на  
                                // строку имени окна  
    DWORD dwStyle,             // стиль окна  
    int x,                     // горизонтальная позиция окна  
    int y,                     // вертикальная позиция окна  
    int nWidth,                // ширина окна  
    int nHeight,               // высота окна  
    HWND hWndParent,           // дескриптор родительского окна  
    HMENU hMenu,               // дескриптор меню или  
                                // идентификатора дочернего окна  
    HANDLE hInstance,          // дескриптор приложения  
    LPVOID lpParam              // указатель на значение,  
                                // передаваемое окну структурой CREATESTRUCT  
);
```

Если функция завершена успешно, то возвращаемое значение является дескриптором нового окна. В случае неуспеха функция возвращает NULL.

Пример:

```
hWnd = CreateWindow(szWindowClass,    //указатель на  
                    //имя класса  
    szTitle,                //указатель на имя окна  
    WS_OVERLAPPEDWINDOW,    //стиль окна  
    GetSystemMetrics(SM_CXSCREEN)/2-600, //координата  
                                // левого края окна  
    GetSystemMetrics(SM_CYSCREEN)/2-300, //координата  
                                // верхнего края окна  
    GetSystemMetrics(SM_CXSCREEN)/2+55, //ширина окна  
    GetSystemMetrics(SM_CYSCREEN)/2+150, //высота окна  
    NULL,                   //дескриптор окна-родителя  
    NULL,                   //дескриптор меню  
    hInstance,              //дескриптор приложения  
    NULL                    //указатель на структуру инициализации  
);
```

### Главное окно. Дочерние окна

**Главное окно** – окно, которое служит основным согласующим звеном между пользователем и приложением. **Дочернее окно** – ок-

но, имеющее стиль (параметр dwStyle) WS\_CHILD или WS\_CHILDWINDOW. Дочернее окно всегда отображается внутри клиентской области родительского окна.

**Пример:**

```
CreateWindow(szChildWindowClass, szChildTitle,  
WS_CHILDWINDOW, CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, hWnd,  
NULL, hInst, NULL);
```

### ***Отображение окон на экране***

Функция *ShowWindow* устанавливает режим отображения окна.

```
BOOL ShowWindow(  
    HWND hWnd,           // Дескриптор окна  
    int nCmdShow          // Режим отображения окна  
);
```

Если окно видимо, то функция возвращает ненулевое значение. Иначе – возвращает 0.

Функция *UpdateWindow* обновляет клиентскую область заданного окна посредством посылки события WM\_PAINT оконной процедуре.

```
BOOL UpdateWindow(  
    HWND hWnd // Дескриптор окна  
);
```

Если функция выполняется успешно, то возвращаемое значение – не ноль. Иначе функция возвращает 0.

```
typedef struct tagPAINTSTRUCT {  
    HDC hdc;           // Дескриптор контекста устройства  
    BOOL fErase;       // Определяет, очищен ли фон  
    RECT rcPaint;      // Структура, содержащая координаты  
                      // области окна  
    BOOL fRestore;     // Зарезервировано  
    BOOL fIncUpdate;   // Зарезервировано  
    BYTE rgbReserved[32]; // Зарезервировано  
} PAINTSTRUCT;
```

Функция *BeginPaint* подготавливает указанное окно к рисованию и заполняет поля структуры PAINTSTRUCT.

```
HDC BeginPaint(  
    HWND hwnd,           // Дескриптор окна  
    LPPAINTSTRUCT lpPaint // Указатель на структуру  
                      // PAINTSTRUCT  
);
```

Если функция завершается успешно, то она возвращает указатель на контекст дисплея, иначе – возвращает NULL.

Функция *GetClientRect* определяет, успешно ли выделена клиентская область. Возвращает TRUE или FALSE.

Функция *EndPaint* отмечает окончание рисования указанного окна.

### ***Управление положением и размерами окон***

Управление положением и размерами окна (в пикселах) на этапе его создания осуществляется параметрами функции *CreateWindow*:

```
int x,           // Горизонтальная позиция
                // левого верхнего угла окна
int y           // Вертикальная позиция левого верхнего угла
int nWidth,     // Ширина окна
int nHeight,    // Высота окна
```

Для управления положением и размерами окна во время выполнения программы можно использовать функцию *SetWindowPos*:

```
BOOL SetWindowPos (
    HWND hWnd,           // Дескриптор окна
    HWND hWndInsertAfter, // Порядок расположения окон
    int X,               // Горизонтальная позиция
    int Y,               // Вертикальная позиция
    int cx,              // Ширина окна
    int cy,              // Высота окна
    UINT uFlags           // Флаг позиции окна
);
```

### ***Установка фона окна***

Фон окна осуществляется через определение дескриптора кисти фона окна при регистрации класса окна:

```
wcex.hbrBackground=CreateSolidBrush( RGB(0,150,40) );
```

В процессе работы приложения можно узнать либо установить фон функцией

```
DWORD (Get)SetClassLong( HWND hWnd, int nIndex, LONG dwNewLong );
```

Для этого поле `nIndex` устанавливают в значение `GCL_HBRBACKGROUND`.

## ***Прикрепление курсора к приложению. Создание курсора***

Курсор задается при регистрации класса окна. Для этого используется функция *LoadCursor*:

```
HCURSOR LoadCursor(  
    HINSTANCE hInstance,    // Дескриптор приложения  
    LPCTSTR lpCursorName    // Строка, определяющая курсор  
);
```

Создать значок курсора можно в среде Visual C++, создав ресурс курсора. Также для установления курсора можно использовать функцию *SetCursor*.

## ***Цикл обработки сообщений***

Это программный цикл *while*, который извлекает сообщения типа MSG из очереди сообщений потока и отправляет их в соответствующую оконную функцию. Тело цикла содержит функции обработки сообщений: *GetMessage*, *TranslateAccelerator*, *DispatchMessage*.

Функция *GetMessage* извлекает сообщение из очереди и помещает его в указанную структуру типа MSG.

```
BOOL GetMessage(  
    LPMSG lpmsg,    // Адрес структуры сообщения  
    HWND hWnd,      // Дескриптор окна  
    // Если hWnd == NULL, то функция  
    // извлекает сообщение от любого окна,  
    // относящегося к данному потоку  
    UINT wMsgFilterMin, // Первое сообщение  
    UINT wMsgFilterMax  // Последнее сообщение  
);
```

Структура типа MSG:

```
typedef struct tagMSG { // msg  
    HWND  hwnd;    // Дескриптор окна  
    UINT  message;  // Определяет номер сообщения  
    WPARAM wParam; // Дополнительная информация о сообщении  
    LPARAM lParam;  // Дополнительная информация о сообщении  
    DWORD time;     // Определяет время послыки сообщения  
    POINT pt;       // Определяет позицию курсора, в координатах  
    // экрана, когда сообщение было послано  
} MSG;
```

Функция *TranslateAccelerator* обрабатывает сообщения от акселераторных клавиш для команд меню, преобразует сообщения WM\_KEYDOWN и WM\_SYSKEYDOWN в сообщения WM\_COMMAND и WM\_SYSCOMMAND соответственно.

```
int TranslateAccelerator(  
    HWND hWnd,           // Дескриптор окна - назначения  
    HACCEL hAccTable,    // Указатель на таблицу акселератора  
    LPMMSG lpmsg          // Адрес структуры сообщения  
);
```

Функция *TranslateMessage* преобразует сообщение в виртуальной форме в символьное сообщение. Это символьное сообщение посылается в очередь сообщений потока. В случае успеха функция возвращает ненулевое значение.

```
BOOL TranslateMessage(  
    CONST MSG *lpmsg      // Адрес структуры сообщения  
);
```

Функция *DispatchMessage* отправляет сообщение оконной функции. Возвращаемое значение определяет значение, которое возвращается оконной функцией (чаще всего оно игнорируется).

```
LONG DispatchMessage(  
    CONST MSG *lpmsg      // Указатель на структуру сообщения  
);
```

### ***Оконная функция. Назначение, параметры, обработка сообщений***

***Оконная функция*** – функция, вызываемая операционной системой, которая контролирует внешний вид и поведение ассоциированного с ней окна. Функция принимает и обрабатывает все сообщения к этому окну.

```
LRESULT CALLBACK WindowProc(  
    HWND hwnd,           // Дескриптор окна  
    UINT msg,            // Идентификатор сообщения  
    WPARAM wParam,       // Первый параметр сообщения  
    LPARAM lParam        // Второй параметр сообщения  
);
```

Возвращаемое значение определяется посланным сообщением.

Функция *CallWindowProc* передаёт сообщение указанной оконной функции:

```

LRESULT CallWindowProc(
    WNDPROC lpPrevWndFunc,    // Указатель на
                               // предыдущую процедуру
    HWND hWnd,               // Дескриптор окна
    UINT msg,                 // Сообщение
    WPARAM wParam,           // Первый параметр сообщения
    LPARAM lParam             // Второй параметр сообщения
);

```

Функция *DefWindowProc* осуществляет обработку сообщения по умолчанию.

```

LRESULT DefWindowProc(
    HWND hWnd,               // Дескриптор окна
    UINT msg,                 // Сообщение
    WPARAM wParam,           // Первый параметр сообщения
    LPARAM lParam             // Второй параметр сообщения
);

```

### ***Обновление оконной области на экране***

Для обновления оконной области посылается сообщение WM\_PAINT:

```

switch (message) {
    case WM_PAINT:
        hdc = BeginPaint(hWnd, &ps); //получаем дескриптор
                                     //контекста устройства
        RECT rt; // TODO: Add any drawing code here
        GetClientRect(hWnd, &rt);    //Получаем клиентскую
                                     //область
        EndPaint(hWnd, &ps);         //Завершаем рисование
        break;
}

```

Так как сообщение WM\_PAINT обладает наименьшим приоритетом по сравнению с остальными сообщениями, то для немедленной перерисовки рабочей области окна надо вызвать функцию *UpdateWindow(hWnd)*.

Обработка сообщений производится с помощью конструкции

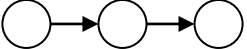
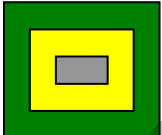
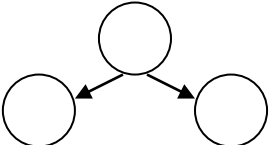

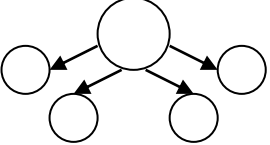
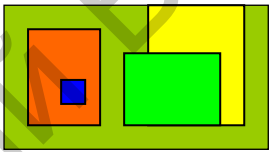

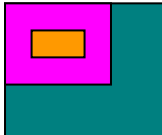

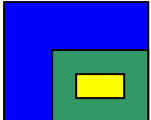
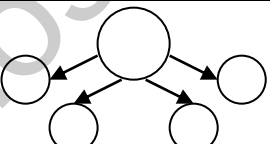
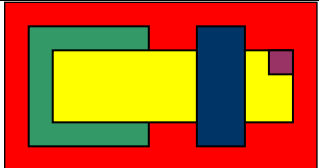

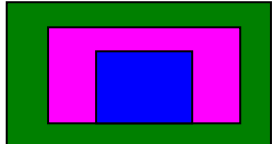
```

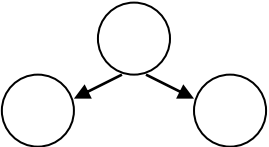

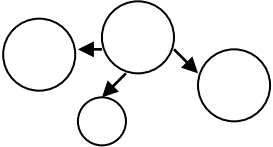
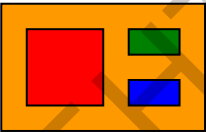
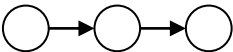

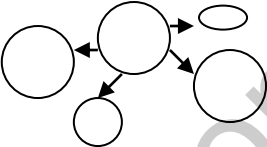
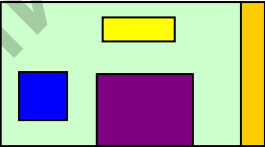
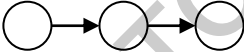
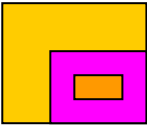

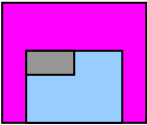
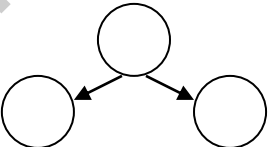
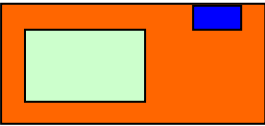
switch (message) {
    case WM_CREATE: ... break;
    case WM_DESTROY: ... break;
    case WM_CHAR: ... break;
    case WM_DESTROY:
        .....
    default: ...
    return DefWindowProc(hWnd, message, wParam, lParam);
}

```



## Варианты заданий к лабораторной работе № 1

Вариант	Схема подчинения окон	Начальное положение окон
1	 <div style="border: 1px solid black; padding: 5px; display: inline-block;">C0   c1   c2</div>	
2		
3		
4	 <div style="border: 1px solid black; padding: 5px; display: inline-block;">C0   c1   c2</div>	
5	 <div style="border: 1px solid black; padding: 5px; display: inline-block;">C0   c1   c2</div>	
6		
7	 <div style="border: 1px solid black; padding: 5px; display: inline-block;">C0   c1   c2</div>	

Вариант	Схема подчинения окон	Начальное положение окон
8		
9		
10		
11		
12	 <div data-bbox="311 925 509 981">C0 c1 c2</div>	
13		
14		

## *Лабораторная работа № 2*

### **ДИАЛоговые ОКНА**

**Цель работы:** изучить вопросы проектирования и создания модальных и немодальных диалоговых панелей на базе Win32/64 API.

#### **Изучаемые вопросы**

1. Назначение и классификация диалоговых окон.
2. Создание и отображение диалогового окна на экране.
3. Шаблон, ресурсы диалогового окна.
4. Задание размеров диалогового окна.
5. Управляющие элементы диалогового окна.
6. Обработка сообщений от элементов управления диалогового окна.
7. Сообщения работы с модальными и немодальными диалоговыми окнами.
8. Обмен информации и проверка корректности вводимых данных.
9. Элементы управления:
  - 9.1. Поле для ввода. Кнопка. Список. Комбинированный список. Слайдер.
  - 9.2. Сообщения к элементам управления SendMessage, SendDlgItemMessage.
  - 9.3. Обработка сообщений от элементов диалогового окна.
  - 9.4. Корректность вводимых данных.
10. Окна сообщений.
11. Стандартные диалоги.

#### **Постановка задачи**

Создать приложение (рис. 2.1), использующее модальный, системный модальный, немодальный и стандартный диалоги и элементы управления типа кнопка, меню, список, движок, поле для ввода.

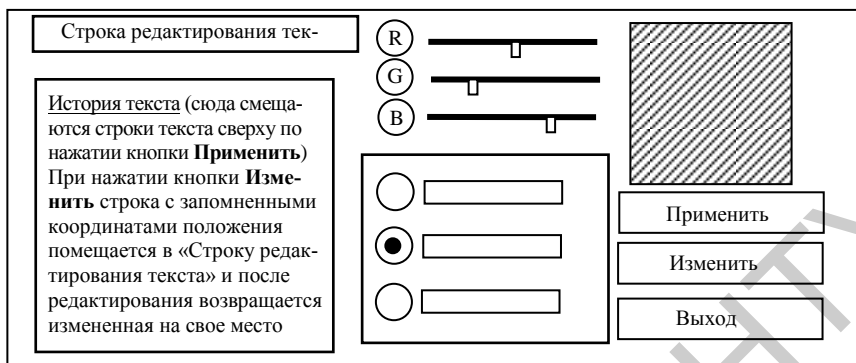


Рис. 2.1. Пример диалогового окна

## Теоретические сведения

### *Назначение и классификация диалоговых окон*

Windows предоставляет много предопределенных или **стандартных** диалоговых окон, которые поддерживают команды типа Открыть файл (File Open) и Печатать файл (File Print). Прикладные программы, которые используют эти команды, должны использовать эти стандартные диалоговые окна для стандартизации интерфейса пользователя.

Существует два типа окон диалогов – модальное и немодальное. **Модальное диалоговое окно** (modal dialog box) требует, чтобы пользователь закрыл диалоговое окно для продолжения работы в прикладной программе. Приложения используют модальные блоки диалога вместе с командами, которые требуют дополнительной информации прежде, чем они могут корректно выполняться.

**Немодальное диалоговое окно** (modeless dialog box) позволяет пользователю передавать фокус ввода любому окну приложения без закрытия блока диалога.

Модальные диалоговые окна более просты для управления, чем немодальные блоки диалога, потому что они создаются, исполняют свою задачу и разрушаются вызовом единственной функции.

**Окно сообщения** (message box) – специальное системное модальное диалоговое окно, которое прикладная программа может использовать, чтобы показывать сообщения и приглашение для обычного ввода данных. Окно сообщения обычно содержит тексто-

вое сообщение и одну или несколько кнопок. Прикладная программа создает окно сообщения, используя функцию `MessageBox` или `MessageBoxEx`, определяя текст, число и типы кнопок для показа. Функция `MessageBoxEx` позволяет также установить язык, который надо использовать для текста любых предопределенных командных кнопок в окне сообщения.

### ***Создание и отображение диалогового окна на экране***

Чтобы создать или модальное, или немодальное диалоговое окно, в прикладной программе надо создать шаблон диалога, для описания стиля, содержания и геометрических параметров диалогового окна, а также код *процедуры диалогового окна*.

**Шаблон диалогового окна** (dialog box template) – бинарное описание блока диалога и элементов управления, которое оно содержит. Разработчик может создать этот шаблон как ресурс, который будет загружен из исполняемого файла прикладной программы, или создать его в памяти, пока выполняется прикладная программа.

**Процедура диалогового окна** (dialog box procedure) – определяемая программой функция повторного вызова, которую Windows вызывает, когда операционная система получает ввод данных для диалогового окна или задачу для выполнения в блоке диалога.

Прикладная программа обычно создает диалоговое окно, используя либо функцию `DialogBox`, либо `CreateDialog`. Функция `DialogBox` создает модальное диалоговое окно; `CreateDialog` создает немодальный блок диалога. Эти две функции загружают шаблон блока диалога из исполняемого файла приложения и создают «выскакивающее» окно, которое соответствует технической спецификации шаблона.

Макрокоманда `CreateDialog` создает немодальное диалоговое окно из ресурса шаблона блока диалога. Макрокоманда `CreateDialog` использует функцию `CreateDialogParam`.

```
HWND CreateDialog
(
    HINSTANCE hInstance,          // дескриптор экземпляра
                                  // прикладной программы
    LPCTSTR lpTemplate,           // идентифицирует название
                                  // шаблона блока диалога
    HWND hWndParent,              // дескриптор окна владельца
    DLGPROC lpDialogFunc          // указатель на процедуру
                                  // диалогового окна
);
```

Параметры:

`hInstance` – идентифицирует экземпляр модуля, исполняемый файл которого содержит шаблон диалогового окна;

`lpTemplate` – идентифицирует шаблон диалогового окна. Этот параметр является указателем или на строку символов с нуль-терминатором в конце, которая определяет название шаблона блока диалога, или на целочисленное значение, которое определяет идентификатор ресурса шаблона диалогового окна. Если параметр устанавливает идентификатор ресурса, его старшее слово должно быть нулевое, а младшее – содержать идентификатор. Можно использовать макрокоманду `MAKEINTRESOURCE`, чтобы создать это значение;

`hWndParent` – дескриптор окна владельца;

`lpDialogFunc` – указатель на процедуру блока диалога.

Если функция завершает работу успешно, возвращаемое значение – дескриптор диалогового окна. Если функция терпит неудачу, возвращаемое значение `NULL`.

Макрокоманда *DialogBox* создает модальное диалоговое окно из ресурса шаблона блока диалога. *DialogBox* не возвращает управления до тех пор, пока заданная функция обратного вызова не прекратит работу модального диалогового окна путем вызова функции *EndDialog*. Макрокоманда *DialogBox* использует функцию *DialogBoxParam*.

```
int DialogBox
(
    HINSTANCE hInstance,    //дескриптор экземпляра программы
    LPCTSTR lpTemplate,     //идентифицирует шаблон
                           //диалогового окна
    HWND hWndParent,       //дескриптор окна владельца
    DLGPROC lpDialogFunc   //указатель на процедуру
                           //блока диалога
);
```

Назначение параметров аналогично параметрам `CreateDialog`.

Если функция завершает свою работу успешно, возвращаемое значение является параметром `nResult` при вызове функции *EndDialog*, используемой для завершения работы блока диалога. Если функция потерпела неудачу, возвращаемое значение – минус 1.

Прикладные программы не имеют прямого доступа к предопределенному в Windows классу окна или *оконной процедуре*, но они могут использовать шаблон блока диалога и процедуру диалогового окна, чтобы изменить его стиль и поведение.

Пример:

**WinAPI:**

```
case IDM_OPTIONS:
    DialogBox(hInst, (LPCTSTR)IDD_OPTIONS_DIALOG, hWnd,
(DLGPROC)Options);
    InvalidateRect(hWnd, &rt, true);
    break;
```

**MFC:**

```
COptionsDialog* optionsDlg = new COptionsDialog();

optionsDlg->Create(IDD_OPTIONS, pFrame);
optionsDlg->Initialize(&(pFrame->m_wndView));
optionsDlg->ShowWindow(SW_SHOW);
```

### ***Шаблон, ресурсы диалогового окна***

Шаблон диалогового окна (dialog box template) – двоичные данные, которые описывают блок диалога, определяя высоту, ширину, стиль и содержащиеся в нем элементы управления. Чтобы создать диалоговое окно, Windows или загружает шаблон блока диалога из ресурса исполняемого файла приложения, или использует шаблон, передав его в глобальную память прикладной программы. И в том и в другом случае прикладная программа должна получить шаблон, когда создается диалоговое окно.

Разработчик, создавая ресурсы шаблонов, использует компилятор ресурсов или редактор диалогового окна. Компилятор ресурсов преобразует текстовое описание ресурса в двоичный код, а редактор сохраняет диалоговое окно как ресурс в двоичном коде в исполняемом файле или динамической библиотеке.

Чтобы создать диалоговое окно без использования ресурсов шаблонов, требуется создать шаблон в памяти и переслать его в функцию *CreateDialogIndirectParam* или *DialogBoxIndirectParam* или в макрокоманду *CreateDialogIndirect* или *DialogBoxIndirect*.

Можно создать шаблон в памяти, размещая его в глобальной памяти и заполняя его стандартным или расширенным заголовком и определениями элементов управления. Шаблон в памяти по форме и содержанию аналогичен шаблону ресурса. Многие прикладные программы, которые используют шаблоны в памяти, сначала используют функцию *LoadResource*, чтобы загрузить ресурс шаблона в память, а затем модифицируют загруженный ресурс, чтобы создать новый шаблон в памяти.

Шаблон диалогового окна в памяти состоит из заголовка, описывающего диалоговое окно, который сопровождается одним или большим числом дополнительных блоков данных, описывающих каждый из элементов управления в блоке диалога. Шаблон может использовать или стандартный, или расширенный формат. В стандартном шаблоне заголовочный файл – это структура `DLGTEMPLATE`, сопровождаемая дополнительными массивами переменной длины. Данные для каждого элемента управления состоят из структуры `DLGITEMTEMPLATE`, сопровождаемой дополнительными массивами переменной длины. В расширенном шаблоне диалогового окна заголовочный файл использует формат `DLGTEMPLATEEX` и определения элементов управления, использующие формат `DLGITEMTEMPLATEEX`.

Чтобы отличить стандартный шаблон от расширенного, следует проверить первые 16 битов шаблона диалогового окна. В расширенном шаблоне первое слово (`WORD`). – `0xFFFF`; любое другое значение показывает, что это стандартный шаблон.

Если создается шаблон блока диалога в памяти, то необходимо гарантировать, что каждый элемент управления `DLGITEMTEMPLATE` или `DLGITEMTEMPLATEEX` выровнен по границе ДВОЙНОГО СЛОВА (`DWORD`). Кроме того, любые данные создания, которые следуют за определением элемента управления, должны быть выровнены по границе ДВОЙНОГО СЛОВА (`DWORD`). Все другие массивы переменной длины в шаблоне диалогового окна должны быть выровнены по границам СЛОВА (`WORD`).



### *Заголовок шаблона*

В обоих шаблонах диалогового окна, стандартном и расширенном, заголовок включает в себя следующую общую информацию:

1) положение и размеры (габариты) диалогового окна в специальных единицах;

2) стили окна и блока диалога для диалогового окна;

3) число элементов управления диалогового окна. Это значение обуславливается числом элементов управления `DLGITEMTEMPLATE` или `DLGITEMTEMPLATEEX`, которые определены в шаблоне;

4) необязательный ресурс меню для диалогового окна. Шаблон может указывать, что блок диалога не имеет меню или он может установить значения по порядку или строку Unicode с нуль-терминатором в конце, которые идентифицируют ресурс меню в исполняемом файле;

5) класс диалогового окна. Это может быть или предопределенный класс диалогового окна, или значения по порядку, или строка Unicode с нуль-терминатором в конце, которые идентифицируют зарегистрированный класс окна;

6) строку Unicode с нуль-терминатором в конце, которая определяет заголовок окна блока диалога. Если строка пустая, то поле заголовка диалогового окна не заполняется. Если в блоке диалога не определен стиль `WS_CAPTION`, система устанавливает заголовок, определенный в строке, но не показывает его;

7) если диалоговое окно имеет стиль `DS_SETFONT`, заголовок устанавливает размер в пунктах и название шрифта, который используется для текста в рабочей области и элементах управления блока диалога.

В расширенном шаблоне заголовок `DLGTEMPLATEEX` определяет к тому же следующую дополнительную информацию:

1) идентификатор контекста справки, который идентифицирует окно блока диалога, когда система посылает сообщение `WM_HELP`;

2) если в диалоговом окне определен стиль `DS_SETFONT`, заголовок устанавливает толщину шрифта и определяет, шрифт курсивным (*italic*).

## ***Определения элементов управления***

Нижеследующий шаблон заголовка – это один или большее количество определений элементов управления, которые описывают элементы управления диалогового окна. И в стандартном и в расширенном шаблоне заголовок блока диалога имеет элемент, который указывает число элементов управления, определяемых в шаблоне. В стандартном шаблоне каждое определение элемента управления состоит из структуры `DLGITEMTEMPLATE`, сопровождаемой дополнительными массивами переменной длины. В расширенном шаблоне определения элемента управления используют формат `DLGITEMTEMPLATEEX`.

И в стандартном и в расширенном шаблоне определение элемента управления включает в себя следующую информацию:

- 1) размещение и габариты элемента управления;
- 2) стили окна и элемента управления для средств управления;
- 3) идентификатор элемента управления;
- 4) класс окна элемента управления. Это могут быть или порядковые значения предопределенного класса системы, или строка Unicode с нуль-терминатором на конце, которая определяет имя зарегистрированного класса окна;
- 5) строка Unicode с нуль-терминатором на конце, которая определяет начальный текст элемента управления, или перечисление значений, которые идентифицируют ресурс, тип пиктограммы в исполняемом файле;
- 6) необязательный переменной длины блок данных создания. Когда система создает элемент управления, она передает указатель на эти данные в параметре `IParam` сообщения `WM_CREATE`, которое передается в элемент управления.

В расширенном шаблоне определение элемента управления также выполняет идентификатор справочного контекста, который идентифицирует элемент управления, когда система посылает сообщение `WM_HELP`.

## ***Задание размеров диалогового окна***

Шаблон каждого диалогового окна содержит размеры, которые устанавливают позицию, ширину и высоту блока диалога и элементов управления, которые он содержит. Это аппаратно-независимые

размеры, так что прикладная программа может использовать единственный шаблон для создания одного и того же диалогового окна для всех типов устройств изображения. Это гарантирует, что блок диалога будет иметь те же самые пропорции и внешний вид на всех экранах, несмотря на различное разрешение и отношение сторон между экранами.

Размеры диалогового окна устанавливаются в базовых единицах диалога. Одна единица по горизонтали эквивалентна одной четверти средней ширины символа системного шрифта. Одна единица по вертикали эквивалентна одной восьмой средней высоты символа системного шрифта. Прикладная программа может извлечь информацию о числе пикселей на базовую единицу измерения для текущего изображения при помощи функции *GetDialogBaseUnits*. Приложение может преобразовать размеры из базовых единиц диалогового окна в пиксели, используя функцию *MapDialogRect*.

Шаблон должен устанавливать начальные координаты верхнего левого угла диалогового окна. Обычно это координаты относительно верхнего левого угла рабочей области окна владельца. Когда шаблон устанавливает стиль *DS\_ABSALIGN* или у диалогового окна нет владельца, позиция определяется относительно верхнего левого угла экрана. Windows устанавливает эту первоначальную позицию, когда создает диалоговое окно, но дает возможность прикладной программе регулировать позицию блока диалога перед его показом. Например, приложение может извлечь габариты окна владельца, вычислить новую позицию, в которую помещает диалоговое окно в окне владельца, а затем установить позицию, используя функцию *SetWindowPos*.

**Пример:**

```
HWND hwndOwner;  
RECT rc, rcDlg, rcOwner;  
case WM_INITDIALOG:  
  
    if ((hwndOwner = GetParent(hDlg)) == NULL)  
    {  
        hwndOwner = GetDesktopWindow();  
    }  
  
    GetWindowRect(hwndOwner, &rcOwner);  
    GetWindowRect(hDlg, &rcDlg);  
    CopyRect(&rc, &rcOwner);
```

```

OffsetRect(&rcDlg, -rcDlg.left, -rcDlg.top);
OffsetRect(&rc, -rc.left, -rc.top);
OffsetRect(&rc, -rcDlg.right, -rcDlg.bottom);

SetWindowPos (hDlg,
    HWND_TOP,
    rcOwner.left + (rc.right / 2),
    rcOwner.top + (rc.bottom / 2),
    0, 0,          // ignores size arguments
    SWP_NOSIZE);

return TRUE;

```

### ***Управляющие элементы диалогового окна***

Windows предоставляет множество функций, сообщений и предопределенных элементов управления, которые помогают создавать и управлять диалоговыми окнами, таким образом облегчая процесс разработки интерфейса пользователя для прикладной программы.

Шаблон устанавливает позицию, ширину, высоту, стиль, идентификаторы и класс окна для каждого элемента управления в диалоговом окне. Windows создает каждый элемент управления путем передачи их данных в функцию *CreateWindowEx*. Элементы управления создаются по порядку, в котором они установлены в шаблоне. Шаблон должен определить соответствующее число, тип и порядок элементов управления, чтобы гарантировать, что пользователь может сделать ввод необходимых данных для завершения команды, связанной с диалоговым окном.

Для каждого элемента управления шаблон устанавливает значения стиля, которые определяют внешний вид и действие элемента управления. Каждый элемент управления – это дочернее окно, и поэтому должен иметь стиль *WS\_CHILD*. Чтобы гарантировать, что элемент управления видимый, когда отображается диалоговое окно, каждый элемент управления должен иметь также стиль *WS\_VISIBLE*. Другие обычно используемые стили окна – это *WS\_BORDER* для элементов управления, которые не обязательно имеют рамки, *WS\_DISABLED* для элементов управления, которые должны быть отключены, когда создается первоначальное диалоговое окно и *WS\_TABSTOP* и *WS\_GROUP* – для элементов управления, к которым можно обращаться, используя клавиатуру. Стили *WS\_TABSTOP* и *WS\_GROUP* используются совместно с диалоговым интерфейсом клавиатуры, который будет описан позже в этом разделе.

Шаблон может также установить стили элементов управления, специфические для класса элементов управления. Например, шаблон, который определяет кнопку управления, должен дать кнопке управления стиль, такой как `BS_PUSHBUTTON` или `BS_CHECKBOX`. Windows передает стиль элемента управления в управляющую окно процедуру через средство сообщения `WM_CREATE`, разрешая процедуре приспосабливать внешний вид и работу элемента управления.

Windows преобразует значения позиции координат, ширину и высоту из базовых единиц диалога в пиксели, перед передачей их в *CreateWindowEx*. Когда Windows создает элемент управления, он определяет диалоговое окно как родительское. Это означает, что Windows всегда воспринимает координаты расположения элемента управления как рабочие координаты относительно верхнего левого угла рабочей области блока диалога.

Шаблон определяет класс окна для каждого элемента управления. Обычно диалоговое окно содержит элементы управления, принадлежащие предопределенным элементам управления класса окна, таким как кнопка и поле редактирования класса окна. В этом случае шаблон определяет класс окна путем присваивания соответствующего предопределенного значения атому класса. Когда диалоговое окно содержит элемент управления, принадлежащий классу пользовательских элементов управления окна, шаблон дает название этого зарегистрированного класса окна или значение атома, в настоящее время связанного с этим названием.

Каждый элемент управления в диалоговом окне должен иметь уникальный идентификатор для отличия его от других элементов управления. Элементы управления передают информацию процедуре блока диалога посредством сообщения `WM_COMMAND`, так что идентификаторы элементов управления необходимы для процедуры, чтобы различать, какой элемент управления передал данное сообщение. Единственным исключением из этого правила являются идентификаторы статических элементов управления. Статические элементы управления не требуют уникального идентификатора, так как они не передают сообщения `WM_COMMAND`.

Чтобы разрешить пользователю закрыть диалоговое окно, шаблон должен установить по крайней мере одну командную кнопку и присвоить ей идентификатор управления `IDCANCEL`. Чтобы разрешить поль-

зователю выбор между завершающей и отменяющей командами, связанными с диалоговым окном, шаблон должен установить две командные кнопки с надписями ОК и Отменить (Cancel) и управляющими идентификаторами IDOK и IDCANCEL соответственно.

Шаблон также устанавливает необязательный текст и данные для создания элемента управления. Текст обычно предназначен для обозначения кнопок управления или установки начального содержания статического текстового элемента управления. Данные создания – это один или большее количество байтов данных, которые Windows передает оконной процедуре элемента управления, когда элемент управления создается, более полезные для элемента управления, который требует дополнительной информации о своем начальном содержании или стиле, чем это дается другими данными. Например, прикладная программа может использовать данные создания, чтобы установить начальные параметры и диапазон управления полосой прокрутки.

### ***Обработка сообщений от элементов управления диалогового окна***

Оповещающие сообщения элемента управления edit:

- EM\_CANUNDOEM\_CHARFROMPOS, EM\_EMPTYUNDOBUFFER,
- EM\_FMTLINES, EM\_GETCUEBANNER, EM\_GETFIRSTVISIBLELINE,
- EM\_GETHANDLE, EM\_GETTIMESTATUS, EM\_GETLIMITTEXT,
- EM\_GETLINE, EM\_GETLINECOUNT, EM\_GETMARGINS.

### ***Сообщения работы с модальными и немодальными диалоговыми окнами***

При создании модального диалогового окна Windows делает его активным окном. Диалоговое окно остается активным до тех пор, пока процедура блока диалога не вызовет функцию *EndDialog* или пока Windows не активизирует окно в другой прикладной программе. Как только Windows создаст модальное диалоговое окно, она посылает сообщение WM\_CANCELMODE окну (если оно есть), которое в настоящее время захватило ввод данных от мыши. Прикладная программа, которая принимает это сообщение, должна освободить захваченную мышь так, чтобы пользователь мог переместить мышь в модальное диалоговое окно. Поскольку Windows

отключает окно владельца, весь ввод данных от мыши будет потерян, если владелец не выполняет требование освободить мышь после приема этого сообщения.

Чтобы обрабатывать сообщения для **модального** диалогового окна, Windows запускает его собственный цикл сообщений, принимая временное управление очередью сообщений для всей прикладной программы. Когда Windows извлекает сообщение, которое явно не для блока диалога, она (ОС) посылает сообщение соответствующему окну. Если оперативная система извлекает сообщение WM\_QUIT, то она передает сообщение назад в очередь сообщения прикладной программы так, чтобы главный цикл сообщений прикладной программы мог в конечном счете отыскать это сообщение.

**Немодальное** диалоговое окно не блокирует окно владельца, не передает какие-либо сообщения для него. Когда создается блок диалога, Windows делает его активным окном, однако пользователь или прикладная программа могут в любое время заменить активное окно. Если диалоговое окно становится неактивным, оно остается в Z-последовательности выше окна владельца, даже если окно владельца активное. Прикладная программа ответственна за извлечение и распределение входящих сообщений для диалогового окна. Большинство приложений использует для этого главный цикл сообщений, чтобы дать возможность пользователю перемещаться и выбирать элементы управления, используя клавиатуру как угодно, но прикладная программа должна вызвать функцию *IsDialogMessage*.

Немодальное диалоговое окно не может вернуть значение в прикладную программу, как это делает модальное диалоговое окно, однако процедура блока диалога может передать информацию в окно владельца, используя функцию *SendMessage*.

Прикладная программа перед завершением работы должна разрушить все немодальные диалоговые окна. Она может разрушить немодальный блок диалога, используя функцию *DestroyWindow*. В большинстве случаев процедура диалогового окна вызывает *DestroyWindow* в ответ на ввод пользователем данных, например таких, как выбор кнопки Отменить (Cancel). Если пользователь не закрывает диалоговое окно таким способом, прикладная программа должна вызвать *DestroyWindow*.

Процедура диалогового окна подобна оконной процедуре, в которую Windows посылает сообщения, чтобы процедура, когда она

имеет информацию, задала или завершила выполнение задачи. В отличие от оконной процедуры процедура диалогового окна никогда не вызывает функцию *DefWindowProc*. Вместо этого она возвращает булево значение TRUE, если она обрабатывает сообщение, или FALSE, если она этого не делает.

Параметры процедуры служат тем же самым целям, что и в оконной процедуре, при помощи параметра *hDlg* процедуру принимает дескриптор окна блока диалога.

Большинство процедур диалогового окна обрабатывает сообщение WM\_INITDIALOG и сообщения WM\_COMMAND, передаваемые элементом управления, но обрабатывает немногие, если имеются какие-либо другие сообщения. Если процедура диалогового окна не обрабатывает сообщение, она должна вернуть значение FALSE, чтобы заставить Windows обработать сообщения внутри себя. Единственное исключение из этого правила – сообщение WM\_INITDIALOG. Процедура диалогового окна должна вернуть значение TRUE, чтобы направить в Windows сообщение WM\_INITDIALOG для дальнейшей обработки. В любом случае процедура не должна вызвать *DefWindowProc*.

Оконная процедура для предопределенного класса диалогового окна выполняет обработку по умолчанию для всех сообщений, которые не обрабатывает процедура блока диалога. Когда процедура диалогового окна возвращает значение FALSE для какого-либо сообщения, предопределенная оконная процедура проверяет сообщения и выполняет по умолчанию следующие действия:

- DM\_GETDEFID – можно передать это сообщение в диалоговое окно. Блок диалога возвращает идентификатор заданной по умолчанию командной кнопки элемента управления, если таковая имеется; в противном случае он возвращает нуль;

- DM\_REPOSITION – можно передать это сообщение в диалоговое окно верхнего уровня. Блок диалога переустанавливается самостоятельно, так как он подгоняет себя в пределах области самого главного окна (рабочего стола программы);

- DM\_SETDEFID – можно передать это сообщение в диалоговое окно. Блок диалога устанавливает заданную по умолчанию командную кнопку в элементе управления, определенном идентификатором элемента управления в параметре *wParam*;



– WM\_ACTIVATE – восстанавливает фокус ввода данных в элементе управления, идентифицированном предварительно сохраненным дескриптором, если диалоговое окно активизировано, иначе процедура сохраняет дескриптор элемента управления, имеющего фокус ввода данных;

– WM\_CHARTOITEM – возвращает нуль;

– WM\_CLOSE – посылает уведомительное сообщение BN\_CLICKED в диалоговое окно, устанавливающее IDCANCEL как идентификатор элемента управления. Если блок диалога имеет идентификатор управления IDCANCEL и элемент управления в настоящее время заблокирован, процедура выдает предупреждение и не передает сообщение;

– WM\_COMPAREITEM – возвращает нуль;

– WM\_ERASEBKGDND – заполняет рабочую область диалогового окна, используя кисть, возвращенную сообщением WM\_CTLCOLOLDLG или цветом окна, который задается по умолчанию;

– WM\_GETFONT – возвращает дескриптор определенного программой шрифта диалогового окна;

– WM\_INITDIALOG – возвращает нуль;

– WM\_LBUTTONDOWN – посылает комбинированному блоку, имеющему фокус ввода данных, сообщение CB\_SHOWDROPDOWN, предписывая элементу управления скрыть его раскрывающийся список. Процедура вызывает *DefWindowProc*, чтобы завершить действие по умолчанию;

– WM\_NCDESTROY – освобождает глобальную память, назначенную для средств редактирования в диалоговом окне (применяется в блоках диалога прикладных программа, базирующихся на Windows, в которых установлен стиль DS\_LOCALEEDIT) и освобождает любой определяемый программой шрифт (применяется в диалоговых окнах, в которых установлен стиль DS\_SETFONT). Процедура вызывает функцию *DefWindowProc*, чтобы завершить действие по умолчанию;

– WM\_NCLBUTTONDOWN – посылает комбинированному блоку, имеющему фокус ввода, сообщение CB\_SHOWDROPDOWN, предписывая элементу управления скрыть его раскрывающийся список. Процедура вызывает *DefWindowProc*, чтобы завершить действие по умолчанию;

– WM\_NEXTDLGCTL – устанавливает фокус ввода данных в следующем или предыдущем элементе управления в диалоговом окне, в элементе управления, идентифицированном дескриптором в параметре `wParam`, или в первом элементе управления в диалоговом окне, которое является видимым, не заблокированным и имеет стиль `WS_TABSTOP`. Процедура игнорирует это сообщение, если текущее окно с фокусом ввода данных – не элемент управления;

– WM\_SETFOCUS – устанавливает фокус ввода данных в элемент управления, идентифицированный предварительно сохраненным дескриптором окна элемента управления. Если такого дескриптора не существует, процедура устанавливает фокус ввода данных в первый элемент управления в шаблоне диалогового окна, который является видимым, не заблокированным и имеет стиль `WS_TABSTOP`. Если такого элемента управления не существует, процедура устанавливает фокус ввода данных в первый элемент управления в шаблоне;

– WM\_SHOWWINDOW – если диалоговое окно скрыто, сохраняет дескриптор элемента управления, имеющего фокус ввода данных, затем вызывает *DefWindowProc*, чтобы завершить действие по умолчанию;

– WM\_SYSCOMMAND – если диалоговое окно свернуто, сохраняет дескриптор элемента управления, имеющего фокус ввода данных, затем вызывает *DefWindowProc*, чтобы завершить действие по умолчанию;

– WM\_VKEYTOITEM – возвращает нуль.

Пример:

```
// Getting Related Points
temp1 = GetDlgItemInt(hDlg, IDC_POINTS_EDIT,
&Translated, FALSE);

if ((!Translated) || temp1 > 3000)
{
    MessageBox(hDlg, "Invalid value for Related Points. Value
must be in 0..3000", "Error", MB_OK | MB_ICONERROR);
    SendDlgItemMessage(hDlg, IDC_POINTS_EDIT,
WM_ACTIVATE, WA_ACTIVE, NULL);
    break;
}
ptCount = temp1;
```

На рис. 2.2, 2.3 продемонстрированы примеры диалоговых окон, предлагаемых для разработки в рамках лабораторной работы.

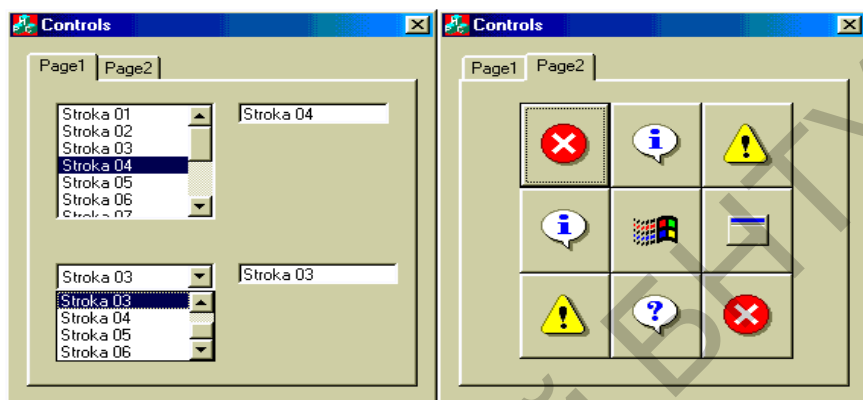


Рис. 2.2. Вкладки с элементами управления

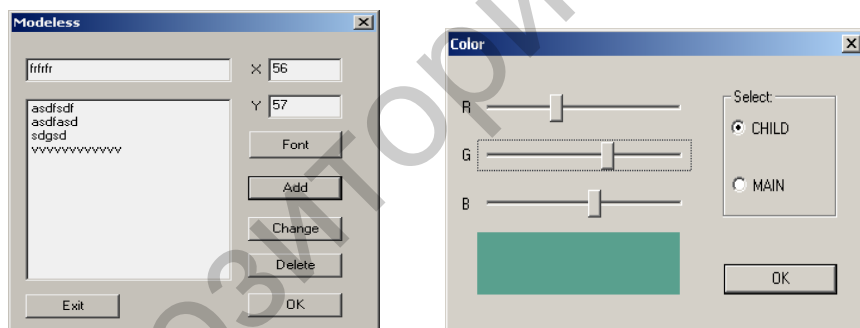


Рис. 2.3. Примеры диалоговых окон

## *Лабораторная работа № 3*

### **ПРОГРАММИРОВАНИЕ ВНЕШНИХ УСТРОЙСТВ**

**Цель работы:** Изучить основы программирования аппаратных устройств: клавиатуры, мыши, таймера.

#### **Изучаемые вопросы**

##### **1. Клавиатура.**

- 1.1. Механизм сообщений от клавиатуры.
- 1.2. Коды OEM, ANSI, ASCII, виртуальные коды.
- 1.3. Виртуальные клавиши.
- 1.4. Символьные сообщения.
- 1.5. Системные сообщения.
- 1.6. Преобразование кодов.
- 1.7. Как отличить нажатия совпадающих клавиш.
- 1.8. Как узнать ввод строчных и прописных букв.
- 1.9. Фокус ввода.

##### **2. Мышь.**

- 2.1. Захват и освобождение окном мыши.
- 2.2. Обработка сообщений от мыши.
- 2.3. Координаты  $x$ ,  $y$ .
- 2.4. Курсор мыши. Управление курсором.
- 2.5. Сообщения мыши в клиентской и неклиентской области

окна.

##### **3. Таймер.**

- 3.1. Установка и снятие таймера.
- 3.2. Функция таймера.

#### ***Постановка задачи***

На базе архитектуры WIN 32 Application создать многооконное приложение, где первое окно отображает информацию по клавиатуре, второе окно отображает информацию по мыши и третье окно отображает информацию по таймеру.

## Теоретические сведения

### *Клавиатура*

При нажатии или отпускании клавиши аппаратные средства клавиатуры генерируют два различных сканкода, которые идентифицируют клавишу. При нажатии клавиш генерируются коды в диапазоне 01–58 Н, а при отжатии – 80 Н и выше. С точки зрения клавиатуры два сканкода – это единственная информация о конкретной клавише.

Значения сканкодов передаются в программную среду, обрабатывающую ввод с клавиатуры. В нашем случае это драйвер клавиатуры Windows.

### *Драйвер клавиатуры Windows*

С началом работы Windows драйвер клавиатуры устанавливает обработчик прерываний для получения сканкодов. Обработчик прерываний – это программа, вызываемая всякий раз при нажатии и отпускании клавиши. Она считывает сканкоды из порта клавиатуры и преобразует их в виртуальные коды клавиши (Virtual keys), совокупность которых называется виртуальной клавиатурой Windows.

Виртуальные коды клавиш – символьные константы этих кодов (макросы) – определены в файле Windows.h и имеют вид **VK\_NAME**. Например, при нажатии клавиши F1 драйвер генерирует виртуальный код **VK\_F1**. Как только драйвер клавиатуры преобразовал сканирующий код клавиши в виртуальный, он вызывает Windows.

Windows помещает сканирующий и виртуальный коды в системный буфер, называемый очередью аппаратных событий.

### *Очередь аппаратных событий*

**Очередь аппаратных событий** – это просто буфер клавиатуры. Он может хранить до 120 событий (т. е. 60 нажатий и 60 отжатий клавиш).

Любой ввод данных в программу, включая ввод с клавиатуры, организован в виде сообщений. Поскольку Windows не прерывает программу, чтобы доставить введенные с клавиатуры данные, эти данные хранятся в буфере клавиатуры.

Содержимое очереди аппаратных сообщений в конечном счете поступает в программу Windows в форме сообщений WM\_KEYDOWN и WM\_KEYUP. Они соответствуют двум сканкодам: нажатия и отпускания клавиши. Программа (оконная процедура окна, имеющего фокус ввода) получает эти сообщения с помощью вызова процедуры GetMessage:

```
while (GetMessage(&msg, NULL, 0,0))
{
    TranslateMessage(&msg); // Разрешить использование
                           //клавиатуры
    DispatchMessage(&msg); // Вернуть управление Windows
}
```

Реальное же значение берется из 4-байтных целых чисел, называемых wParam и lParam, параметров оконной процедуры. Формат этих полей одинаков для обоих сообщений.

wParam содержит виртуальный код нажатой или отпущенной клавиши. Драйвер клавиатуры генерирует его, получая сканирующий код клавиши. Виртуальный код показывает, как драйвер клавиатуры воспринимает события на клавиатуре (нажатие или отпускание), и подключен к машине в контексте Windows.

Параметр lParam разбит на шесть полей и представлен на рис. 3.1.

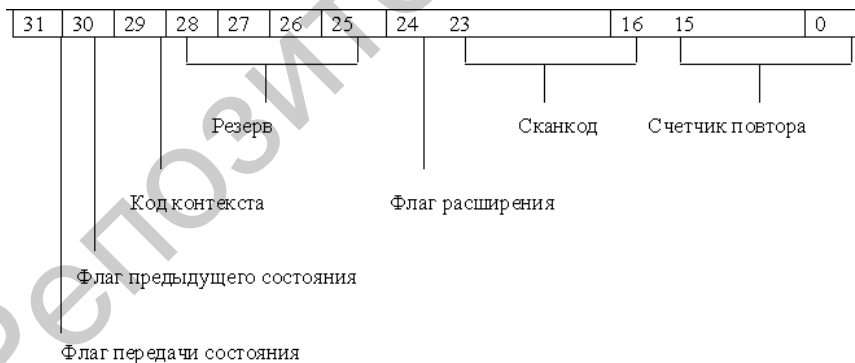


Рис. 3.1. Поля параметра lParam

## Поля параметра *lParam*

*Счетчик повтора.* Конструкция клавиатуры позволяет повторить генерацию сканкода символа, если нажать и удерживать соответствующую клавишу (свойство тайпматика).

Чтобы предотвратить переполнение очереди аппаратных событий такими последовательностями сканкодов, Windows увеличивает *счетчик повтора*, если оказывается, что новое событие клавиатуры совпадает с предыдущим, уже помещенным в очередь.

*Сканкод* – содержит OEM код клавиши клавиатуры. В большинстве программ удобнее пользоваться виртуальными кодами, не зависящими от клавиатуры, а не сканкодами.

Но иногда надо использовать сканкод клавиши, так как требуется отличить нажатие совпадающих клавиш, например, – левой Shift от правой Shift.

*Флаг расширения* – это дополнительная информация о сканкоде. Можно показать, что у расширенной 101-, 102 – клавишной клавиатуры IBM существует клавиша-дублер. Как и сканкод, значение этого поля зависит от аппаратной организации.

*Код контекста.* Код = 1, если нажата клавиша Alt, иначе Код = 0.

*Флаг предыдущего состояния.* Равен 1, если в предыдущем состоянии клавиша была нажата, 0 – в противном случае. Обеспечивает тайпматику.

*Флаг передачи состояния.* 1 – если клавиша отпущена (WM\_KEYUP), 0 – если клавиша нажата (WM\_KEYDOWN).

Из событий WM\_KEYUP и WM\_KEYDOWN программа получает как сканкоды клавиш (*lParam*), так и виртуальные коды (*wParam*). Однако, нажав некоторые клавиши, мы не получим сообщений, так как назначение этих клавиш – не отображать символы, а, скорее, посылать команды.

Нажатие клавиш, приведенных в табл. 3.1, представляется только сообщениями WM\_KEYDOWN и WM\_KEYUP, эти клавиши не имеют ASCII-кодов, поэтому они не генерируют сообщения WM\_CHAR.

Таблица 3.1

## Управляющие клавиши

Клавиша	Назначение
F1–F9, F11–F16	Функциональные клавиши. F10 зарезервирована системой Windows как горячая клавиша выбора меню
Shift, Ctrl, Alt	Клавиши регистра. Alt – WM_SYSKEYDOWN и WM_SYSKEYUP (зарезервирована системой). (Alt + Ctrl) – WM_KEYDOWN, WM_KEYUP
Caps, Lock, NumLock, ScrollLock	Клавиши переключения
Print Screen	Зарезервированная клавиша для копирования содержимого экрана в буфер принтера (если нажата одна клавиша Print Screen) или содержимого активного окна в буфер принтера (Alt + Print Screen)
Insert, Delete, Home, End, PageUp, PageDown	Клавиши редактирования текста. Хотя на расширенной 101-, 102 – клавишной клавиатуре каждой из этих клавиш по две, виртуальные коды каждой пары совпадают. Отличить клавиши можно с помощью флага расширения
Up, Left, Down, Right	Клавиши направления. Парные, но их виртуальные коды равны. Отличить клавишу от своего дублера можно с помощью флага расширения

Чтобы распознать нажатия перечисленных выше клавиш в программе Windows, проверяется значение параметра wParam и сравнивается с соответствующим виртуальным кодом.

Пример. Проверка, была ли нажата клавиша F1?

```
case WM_KEYDOWN:
{
    if (wParam == VK_F1) {} // TRUE - клавиша F1 нажата
    {} // FALSE клавиша F1 не нажата
}
```



Сообщения WM\_KEYDOWN и WM\_KEYUP в основном используются для программирования нажатия/отпускаания несимвольных клавиш.

Необходимо также учитывать разницу между прописными и строчными буквами, поэтому нужно знать и состояние клавиши Shift во время нажатия интересующей клавиши.

Однако нет необходимости проделявать всю эту работу вручную, так как Windows имеет встроенное средство, облегчающее работу с сообщениями. Это процедура TranslateMessage, являющаяся составной частью любого цикла сообщений.

### *Цикл получения сообщений*

Минимальный стандартный цикл сообщений имеет такой вид:

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

Программа GetMessage считывает сообщения из очереди аппаратных событий и из очереди сообщений конкретной программы. Для каждого найденного сообщения вызывается программа TranslateMessage, которая игнорирует все сообщения, кроме WM\_KEYDOWN и WM\_SYSKEYDOWN.

Роль функции TranslateMessage проста. Она получает данные в виде виртуальных кодов из сообщения WM\_KEYDOWN (или WM\_SYSKEYDOWN) и вызывает драйвер клавиатуры для преобразования виртуальных кодов в коды ANSI. Для клавиш, не имеющих кодов ANSI, преобразование не производится. В результате генерируется сообщение WM\_CHAR (или WM\_SYSCHAR), которое размещается в очередь сообщений потока процесса.

В табл. 3.2 и 3.3 приведены сообщения, генерируемые при работе с символьными клавишами.

Таблица 3.2

**Сообщения, генерируемые при вводе строчного символа**

Сообщение	wParam
WM_KEYDOWN	Виртуальный код символа
WM_CHAR	ANSI – код символа
WM_KEYUP	Виртуальный код символа

Таблица 3.3

**Сообщения, генерируемые при вводе прописного символа**

Сообщение	wParam
WM_KEYDOWN	Виртуальный код VK_SHIFT
WM_KEYDOWN	Виртуальный код символа
WM_CHAR	ANSI – код символа
WM_KEYUP	Виртуальный код символа
WM_KEYUP	Виртуальный код VK_SHIFT

В некоторых неанглоязычных странах существуют специальные комбинации клавиш для создания диакритических знаков. Эти комбинации называются мертвыми, так как их нажатие не создает символа, а модифицирует его. Для таких клавиш в соответствии с сообщением WM\_KEYDOWN (WM\_SYSKEYDOWN) программа TranslateMessage генерирует сообщение WM\_DEADCHAR (WM\_SYS-DEADCHAR). Можно игнорировать это сообщение, так как Windows сама создаст нужный символ.

Результатом ввода с клавиатуры является сообщение WM\_CHAR, создаваемое программой TranslateMessage. Затем сообщения WM\_KEYDOWN и WM\_SYSKEYDOWN помещаются программой DispatchMessage в оконную процедуру для обработки.

Для организации надежной обработки ввода символов нужно отфильтровать некоторые сообщения WM\_CHAR, которые были созданы после нажатия «непечатных» символьных клавиш. К таким клавишам относятся, например, BackSpace, Tab и Return. Нажатие этих клавиш, как и других, требует специальной, отличной от обычных символьных клавиш, обработки.

Таблица 3.4

## Коды ANSI

Сочетание клавиш	Код		Описание
	Десятичный	Шестнадцатеричный	
Ctrl + A	10	16	–
Ctrl + G	1 – 7	1 – 7	Непечатные символы
BackSpace	8	8	Клавиша возврата (VK_BACK)
Ctrl + H	8	8	Эмуляция клавиш возврата (VK_BACK)
Tab	9	9	Клавиша табуляции (VK_TAB)
Ctrl + I	9	9	Эмуляция клавиши табуляции (VK_TAB)
Ctrl + J	10	A	Перевод строки
Ctrl + K, Ctrl + L	11 – 12	B – C	Непечатные символы
Return	13	D	Клавиша ввода (VK_RETURN)
Ctrl + M	13	D	Эмуляция ввода (VK_RETURN)
Ctrl + N, Ctrl + Z	15 – 26	E – 1A	Непечатные символы
Esc	27	1B	Клавиша выхода (VK_ESCAPE)

**Оконный объект**

Хотя определенная часть управления клавиатурой выполняется циклом сообщений, основная нагрузка возлагается на соответствующие функции – обработчики оконного объекта – оконную функцию. Можно игнорировать большинство сообщений, сосредоточив внимание на WM\_CHAR и WM\_KEYDOWN (при вводе «непечатных» символов).

В некоторых случаях для обработки ввода с клавиатуры необходимы системные сообщения; в частности, когда активное окно системы пиктограммное, Windows заменяет обычные сообщения клавиатуры соответствующими им системными сообщениями (WM\_SYSKEYDOWN, WM\_SYSKEYUP, WM\_SYSCHAR).

**Оконная процедура умолчания**

Все сообщения, не используемые оконной процедурой, попадают в оконную процедуру умолчания (Default Window procedure): **DefWindowProc(hwnd, message, wParam, lParam)**.

Стандартная оконная процедура игнорирует все обычные сообщения клавиатуры, поэтому оконная процедура, не работающая с сообщениями клавиатуры, может смело пропускать их.

### ***Мышь***

При получении сообщений мыши, связанных с рабочей областью окна, через параметр `wParam` передается значение, позволяющее определить, были ли одновременно с этим нажаты кнопки мыши или клавиши `<Shift>` и `<Ctrl>` клавиатуры. Например, если обработка должна зависеть от состояния клавиш `<Shift>` и `<Ctrl>`, то приложение могло бы воспользоваться следующей логикой:

```
UINT fwKeys = wParam; // состояние кнопок мыши
if(MK_SHIFT & fwKeys)
{
    if(MK_CONTROL & fwKeys)    { }
/* нажаты клавиши <Shift> и <Ctrl> */
    else    { } /* нажата клавиша <Shift> */
}
else
{
    if(MK_CONTROL & fwKeys)    { }
/* нажата клавиша <Ctrl> */
    else    { } /* клавиши <Shift> и <Ctrl> не нажаты */
}
```

Функция *GetKeyState* также может возвращать состояние кнопок мыши или клавиш `<Shift>` и `<Ctrl>`, используя виртуальные коды клавиш `VK_LBUTTON`, `VK_RBUTTON`, `VK_MBUTTON`, `VK_SHIFT` и `VK_CONTROL`. При нажатой кнопке или клавише возвращаемое значение функции *GetKeyState* отрицательно.

### ***Двойной щелчок мыши***

Если необходимо, чтобы оконная процедура получала сообщения двойного щелчка мыши, то следует включить идентификатор **CS\_DBLCLKS** при задании стиля класса окна перед вызовом функции *RegisterClass*, например:

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

Если CS\_DBLCLKS не включить в стиль окна и пользователь в быстром темпе щелкнет левой кнопкой мыши, то оконная процедура получит сообщения о двух одинарных щелчках мыши в следующей последовательности:

```
WM_LBUTTONDOWN, WM_LBUTTONUP, WM_LBUTTONDOWN, WM_LBUTTONUP.
```

### ***Использование таймера***

Хотя драйвер SYSTEM.DRV поддерживает асинхронное прерывание Int 08h, сообщение WM\_TIMER не является асинхронным. Сообщение WM\_TIMER помещается в очередь приложения и обслуживается так же, как и другие сообщения, при этом сообщение WM\_TIMER похоже на WM\_PAINT.

Таймер можно установить/снять в программе, используя функции SetTimer в главной функции или в ветвях обработки сообщений WM\_CREATE/WM\_DESTROY.

Функция установки *SetTimer* ( ) имеет вид

```
SetTimer (hwnd, ID_TIMER, wMsecInterval, NULL),
```

где hwnd – дескриптор окна, функция, которая будет поддерживать обработку WM\_TIMER;

параметр ID\_TIMER – номер таймера, неотрицательная величина;

wMsecInterval – задает интервал в миллисекундах.

Когда оконная функция получает сообщение WM\_TIMER, wParam равен ID\_TIMER и lParam равен нулю.

Ниже приведен пример установки и использования двух таймеров.

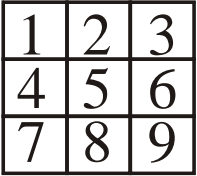
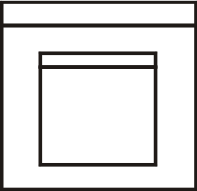
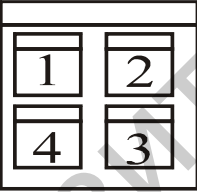

```
# define ID_TIMER_SEC 1
# define ID_TIMER_MIN 2

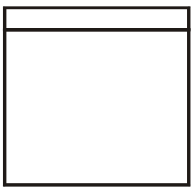
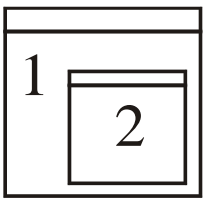
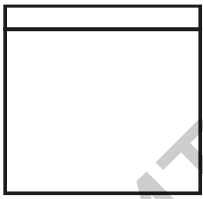
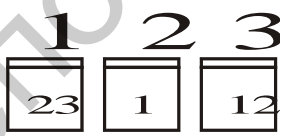
SetTimer (hwnd, ID_TIMER_SEC, 1000, NULL);
SetTimer (hwnd, ID_TIMER_MIN, 60000, NULL);


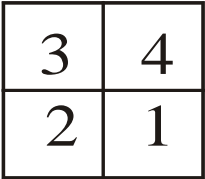
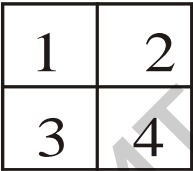
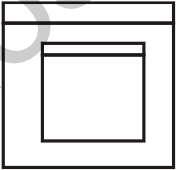
Case WM_TIMER:
    switch (wParam)
    {
        case ID_TIMER_SEC: ..... //код обработчика 1 таймера..
        break;

        ...
        case ID_TIMER_MIN: ..... //код обработчика 2 таймера..
        break;
    }
```

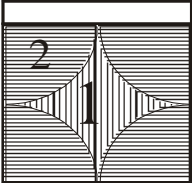
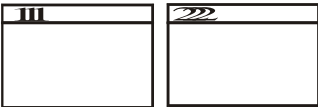
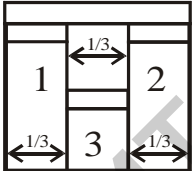
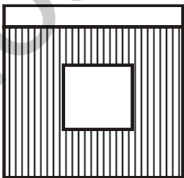
### Варианты заданий к лабораторной работе № 3

Вариант	Рисунок к заданию	Задание
1		<p>1. Через определенный промежуток времени создается 9 окон разных стилей, как показано на рисунке.</p> <p>2. Щелчок мыши в любом окне изменяет его цвет.</p> <p>3. Нажатие клавиш &lt;Alt&gt;+&lt;номер окна&gt; закрывает окно данного номера</p>
2		<p>1. Через определенный промежуток времени в главном окне программы создается дочернее окно, как показано на рисунке.</p> <p>2. Дочернее окно движется внутри главного с помощью клавиш &lt;&lt;-&gt;, &lt;↑&gt;, &lt;-&gt;&gt;, &lt;↓&gt;.</p> <p>3. Щелчок мыши в дочернем окне закрывает его</p>
3		<p>1. По щелчку мыши в главном окне программы появляются четыре окна через <math>\Delta t_1</math> в последовательности 2-3-1-4. Через <math>\Delta t_2</math> окна изменяют свой размер в последовательности 1-2-3-4.</p> <p>2. По нажатию клавиш &lt;Ctrl&gt;+&lt;номер окна&gt; окно этого номера изменяет свой стиль</p>
4		<p>1. Через <math>\Delta t_1</math> создается второе окно.</p> <p>2. Через <math>\Delta t_2</math> фокус ввода переходит от одного окна к другому.</p> <p>3. В окне, имеющем фокус ввода в данный момент, отображается вводимый с клавиатуры символ.</p> <p>4. По щелчку мыши окно очищается</p>

Вариант	Рисунок к заданию	Задание
5		<p>1. Через <math>\Delta t_1</math> окно изменяет стиль, через <math>\Delta t_2</math> окно изменяет цвет.</p> <p>2. Окно двигается по экрану с помощью клавиш <math>\leftarrow</math>, <math>\uparrow</math>, <math>\rightarrow</math>, <math>\downarrow</math> с интервалом 20 пикселей.</p> <p>3. Двойной щелчок любой кнопкой мыши – появляется дочернее окно, щелчок левой кнопкой – это окно изменяет размеры</p>
6		<p>1. Через <math>\Delta t_1</math> создается окно № 2.</p> <p>2. Окна перемещаются клавишами <math>\leftarrow</math>, <math>\uparrow</math>, <math>\rightarrow</math>, <math>\downarrow</math>.  <math>\langle \text{Ctrl} \rangle + \langle 1 \rangle</math> перемещается первое окно, <math>\langle \text{Ctrl} \rangle + \langle 2 \rangle</math> перемещается второе окно.</p> <p>3. Двойной щелчок левой клавишей мыши – окно максимизируется</p>
7		<p>1. Через <math>\Delta t_1</math> окно изменяет положение и размер, через <math>\Delta t_2</math> окно изменяет стиль, через <math>\Delta t_3</math> окно изменяет свой заголовок на фамилию.</p> <p>2. Нажатие на любую клавишу – заголовок становится первоначальным</p> <p>3. Щелчок мышью – стиль становится прежним</p>
8		<p>1. Через <math>\Delta t_1</math> создается окно № 2 и окно № 3.</p> <p>2. Затем по таймеру с промежутками в <math>\Delta t_2</math> окна обмениваются информацией по схеме:  1–2 информация «1», 2–3 информация «12», 3–1 информация «23».</p> <p>3. По щелчку мыши окна изменяют размеры.</p> <p>4. Нажатие клавиш <math>\langle 1 \rangle</math>, <math>\langle 2 \rangle</math> или <math>\langle 3 \rangle</math> закрывает соответствующее окно</p>

Вариант	Рисунок к заданию	Задание
9		<p>1. Через <math>\Delta t_1</math> создаются окна 1–3.</p> <p>2. Через <math>\Delta t_2</math> меняются заголовки окон 1–3.</p> <p>3. Нажатие клавиш <math>\langle \text{Alt} \rangle + \langle \text{номер окна} \rangle</math> закрывает окно данного номера.</p> <p>4. По щелчку мыши окна изменяют положение</p>
10		<p>1. Через <math>\Delta t_1</math> создаются окна, как показано на рисунке, в последовательности 1-2-3-4.</p> <p>2. Через <math>\Delta t_2 &gt; \Delta t_1</math> окна уничтожаются в той же последовательности.</p> <p>3. Нажатие клавиш <math>\langle \text{Alt} \rangle + \langle \text{номер окна} \rangle</math> – блокировка уничтожения окна данного номера</p>
11		<p>1. Через <math>\Delta t_1</math> создаются окна, как показано на рисунке, в последовательности 1-2-3-4.</p> <p>2. Нажатие клавиш <math>\langle \text{Alt} \rangle + \langle \text{номер окна} \rangle</math> передает фокус ввода окну данного номера.</p> <p>3. В окне отображается текст, вводимый с клавиатуры.</p> <p>4. Через <math>\Delta t_2</math> фокус ввода передается от окна к окну по схеме 1-2-3-4</p>
12		<p>1. Через <math>\Delta t_1</math> создается окно внутри главного окна программы.</p> <p>2. Через <math>\Delta t_2</math> оно изменяет свои размеры.</p> <p>3. Нажатие клавиш <math>\langle \text{Alt} \rangle + \langle \leftarrow \rangle</math>, <math>\langle \text{Alt} \rangle + \langle \uparrow \rangle</math>, <math>\langle \text{Alt} \rangle + \langle \rightarrow \rangle</math>, <math>\langle \text{Alt} \rangle + \langle \downarrow \rangle</math> перемещает окно</p>



Вариант	Рисунок к заданию	Задание
13		<ol style="list-style-type: none"> <li>1. Через <math>\Delta t_1</math> создается регион «1».</li> <li>2. Через <math>\Delta t_2</math> создается регион «2».</li> <li>3. По щелчку мыши изменяется стиль окна.</li> <li>4. С клавиатуры вводятся символы, и по нажатию клавиши &lt;Enter&gt; они становятся заголовком окна</li> </ol>
14		<ol style="list-style-type: none"> <li>1. Через <math>\Delta t_1</math> последовательно создаются 2 окна.</li> <li>2. При изменении размеров и положения <math>N</math>-го окна другое окно отображает его координаты и размеры.</li> <li>3. При нажатии клавиши меняется курсор</li> </ol>
15		<ol style="list-style-type: none"> <li>1. Через <math>\Delta t_1</math> главное окно программы случайным образом меняет положение и размер.</li> <li>2. Через <math>\Delta t_2</math> последовательно создаются 3 окна (1-2-3).</li> <li>3. По щелчку мыши высота окна № 3 становится равной высоте двух других окон.</li> <li>4. По нажатию клавиш &lt;Alt&gt;+&lt;номер окна&gt; окно этого номера изменяет свой стиль</li> </ol>
16		<ol style="list-style-type: none"> <li>1. Через <math>\Delta t_1</math> (1 с) в центре окна создается регион, размеры которого равны <math>1/3</math> размеров окна.</li> <li>2. После нажатия клавиши &lt;Enter&gt; их значение выводится на экран.</li> <li>3. Щелчок левой клавишей мыши – регион и надписи исчезают, правой – появляются снова</li> </ol>

Вариант	Рисунок к заданию	Задание
17		<p>1. Через 1 с последовательно создаются окна. Каждое окно имеет свой стиль.</p> <p>2. Через <math>\Delta t_2 &gt; 1</math> с окна в той же последовательности изменяют заголовки.</p> <p>3. По нажатию клавиш <math>\langle \text{Ctrl} \rangle + \langle \text{номер окна} \rangle</math> окну этого номера передается фокус ввода</p>
18		<p>1. Через <math>\Delta t_1</math> в центре экрана создается окно, размеры которого равны <math>\frac{1}{4}</math> размеров экрана.</p> <p>2. Окно двигается по экрану с помощью клавиш <math>\langle \leftarrow \rangle</math>, <math>\langle \uparrow \rangle</math>, <math>\langle \rightarrow \rangle</math>, <math>\langle \downarrow \rangle</math>.</p> <p>3. По левому щелчку мыши в этом окне появляется дочернее окно, по правому щелчку оно пропадает</p>
19		<p>1. Через <math>\Delta t_1</math> создаются два окна.</p> <p>2. По нажатию клавиш <math>\langle \text{Ctrl} \rangle + \langle \text{номер окна} \rangle</math> окну этого номера передается фокус ввода.</p> <p>3. Щелчок мыши очищает окно от введенного текста.</p> <p>4. Из активного окна к другому передается сообщение (текст)</p>
20		<p>1. Через <math>\Delta t_1</math> в главном окне программы создается дочернее окно.</p> <p>2. В главном окне содержится информация о положении клавиши <math>\langle \text{NumLock} \rangle</math>, в дочернем – о <math>\langle \text{CapsLock} \rangle</math>.</p> <p>3. По щелчку мыши окна обмениваются функциями: в главном окне информация о <math>\langle \text{CapsLock} \rangle</math>, в дочернем – о <math>\langle \text{NumLock} \rangle</math></p>

## *Лабораторная работа № 4*

### **РАБОТА С ФАЙЛАМИ**

**Цель работы:** изучить основы работы с двоичным и текстовым файлами на базе WIN32 API.

#### **Изучаемые вопросы**

1. Создание текстового файла. Структура текстового файла.
2. Запись/чтение текстового файла.
3. Создание двоичного файла. Структура двоичного файла.
4. Запись/чтение двоичного файла.
5. Атрибуты файла. Чтение/установка атрибутов файла.
6. Преобразование информации при записи и чтении в/из файла.
7. Внутреннее представление информации разного типа в оперативной и дисковой памяти (файлах).
8. Дампы памяти.

#### **Постановка задачи**

Разработать WIN32 Application с диалогами, которое должно обеспечить:

1. Запись в оба типа файлов данных следующих типов (значение данных задаются через диалоговое окно):

- BYTE, UINT;
- INT32/ INT64;
- WORD, DWORD;
- LONG, BOOL;
- Float; Double;
- Char;

– Строки (в файле должно быть три и более строки).

2. Чтение данных из двоичного файла и их отображение в диалоговом окне.

3. Чтение данных из текстового файла и их отображение в диалоговом окне.

4. Отображение дампов значений данных файлов.

5. Для сохранения файла и его загрузки использовать стандартные диалоги посредством вызова функций `GetOpenFileName()` и `GetSaveFileName()`.

В результате выполнения работы студент должен уметь анализировать кодировку данных в символьном/текстовом и двоичном форматах на основе дампов памяти (файлов). Дамп памяти – отображение байтов памяти (оперативной или дисковой) в шестнадцатеричной системе.

## Теоретические сведения

### Символьные константы

Символьная константа состоит из символа, заключенного в одиночные кавычки (апострофы), как, например, 'x'. Значением символьной константы является численное значение символа в машинном наборе символов (алфавите). Символьные константы считаются данными типа `int`.

Некоторые неграфические символы, одиночная кавычка ' и обратная косая \, могут быть представлены в соответствии табл. 4.1 escape-последовательностей.

Таблица 4.1

### Escape-последовательности

Символ	Код	Обозначение
Символ новой строки	NL(LF)	\n
Горизонтальная табуляция	NT	\t
Вертикальная табуляция	VT	\v
Возврат на шаг	BS	\b
Возврат каретки	CR	\r
Перевод формата	FF	\f
Обратная косая	\	\\
Одиночная кавычка (апостроф)	'	\'
Набор битов	Oddd	\ddd
Набор битов	0xdd	\xdd

Escape-последовательность `\ddd` состоит из обратной косой, за которой следуют одна, две или три **восьмеричные** цифры, задающие значение требуемого символа. Специальным случаем такой конструкции является `\0` (не следует ни одной цифры), задающий пустой символ NULL.

Escape-последовательность `\xddd` состоит из обратной косой, за которой следуют одна, две или три **шестнадцатеричные** цифры, задающие значение требуемого символа. Если следующий за обратной косой символ не является одним из перечисленных, то обратная косая игнорируется.

### ***Правила именования файлов в Win32***

Поскольку Win32 поддерживает несколько файловых систем, все они должны подчиняться неким общим правилам. Имена каталогов и файлов в полном имени файла (pathname) отделяются обратной косой чертой (`\`). Кроме правил формирования полного имени действуют и правила именования каталогов и файлов:

- полное имя файла завершается нулевым символом;
- имена файлов и каталогов не должны содержать разделительного символа (`\`), символов с ANSI-кодами от 0 до 31, а также символов, явно запрещенных в какой-либо файловой системе;
- имена файлов и каталогов могут включать буквы разного регистра, но при поиске файлов и каталогов регистр букв не учитывается. Если файл с именем `ReadMe.Txt` существует, создание нового файла с именем `README.TXT` уже не допускается;
- точка (`.`) идентифицирует текущий каталог. Например, `.\README.TXT` означает, что файл `README.TXT` находится в текущем каталоге;
- две точки (`..`) идентифицируют родительский каталог. Например, `..\README.TXT` означает, что файл `README.TXT` находится в родительском каталоге текущего каталога;
- точка (`.`), используемая как часть имени файла или каталога, считается разделителем компонентов имени. Например, в файле `README.TXT` точка отделяет имя файла от его расширения;
- имена файлов и каталогов не должны содержать некоторых специальных символов вроде `<`, `>`, `:`, `,`, `"` или `]`.

## *Запись информации в файл и чтение информации из файла*

Существуют два режима чтения и записи файлов: синхронный и асинхронный. При синхронном вводе/выводе процесс выборки данных выглядит следующим образом:

1. Программист выделяет в памяти буфер определенного размера.
2. Программист открывает файл, к которому он намерен обращаться.
3. Программист устанавливает указатель файла на то место в файле, где находятся интересующие его данные.
4. Данные из файла считываются в буфер.
5. В буфере производятся определенные действия.
6. Буфер записывается на то же место в файле или добавляется в конец файла.

Естественно, что в реальной ситуации те или иные шаги могут быть опущены.

Прежде чем считывать информацию из файла или записывать информацию в файл, необходимо сначала его открыть. Для этого надо воспользоваться функцией *CreateFile*:

```
HANDLE CreateFile( LPCTSTR lpFileName, // file name
DWORD dwDesiredAccess, // access mode
DWORD dwShareMode, // share mode
LPSECURITY_ATTRIBUTES lpSecurityAttributes, // SD
DWORD dwCreationDisposition, // how to create
DWORD dwFlagsAttributes, // file attributes
HANDLE hTemplateFile // handle to template file
);
```

Четвертый параметр *lpSecurityAttributes* заслуживает особого разговора. Он представляет собой указатель на структуру *SECURITY\_ATTRIBUTES*, которая в файле *winbase.h* описана так:

```
typedef struct _SECURITY_ATTRIBUTES{
DWORD nLength;
LPVOID lpSecurityDescriptor;
BOOL bInheritHandle;
}SECURITY_ATTRIBUTES,*PSECURITY_ATTRIBUTES,
*LPSECURITY_ATTRIBUTES;
```

Если никакой особой защиты файлу не требуется, то в это поле можно занести NULL.

Открыв файл, в конце программы нужно обязательно закрыть его. Для того чтобы закрыть файл, необходимо вызвать функцию *CloseHandle*:

```
BOOL CloseHandle( HANDLE hObject);    // handle to object
```

Чтение данных из файла в синхронном режиме осуществляется функцией *ReadFile*:

```
BOOL ReadFile(
    HANDLE hFile,           // handle to file
    LPVOID lpBuffer,        // data buffer
    DWORD nNumberOfBytesToRead, // number of bytes to read
    LPDWORD lpNumberOfBytesRead, // number of bytes read
    LPOVERLAPPED lpOverlapped // overlapped buffer
);
```

Запись данных в файл в синхронном режиме осуществляется функцией *WriteFile*:

```
BOOL WriteFile( HANDLE hFile, LPCVOID lpBuffer,
    DWORD nNumberOfBytesToWrite,
    LPDWORD lpNumberOfBytesWritten,
    LPOVERLAPPED lpOverlapped );
```

Для того чтобы осуществить асинхронный доступ к файлу, необходимо открыть файл при помощи функции *CreateFile* и указать в аргументе *dwFlagsAttributes* флаг *FILE\_FLAG\_OVERLAPPED*, который укажет, что файл будет открыт именно для асинхронного доступа к данным. Кроме этого, в данном случае в отличие от операций, производимых в синхронном режиме, надо использовать структуру типа *OVERLAPPED*. В файле *winbase.h* эта структура описана следующим образом:

```
typedef struct _OVERLAPPED{
    DWORD Internal;
    DWORD InternalHigh;
    DWORD Offset;
    DWORD OffsetHigh;
    HANDLE hEvent;
}OVERLAPPED, *LPOVERLAPPED;
```

## ***Текстовые файлы***

Текстовые файлы представляют собой набор однобайтовых (ANSI) или двухбайтовых данных (UNICODE), содержащих коды символов, т. е. текстовую информацию.

В оперативной памяти строкой считается последовательность байт, закрытая **нуль** (\0)-символом.

В текстовом файле каждая строка завершается символом возврата каретки (\r) и переводом строки (\n) или 0D0A в шестнадцатеричной кодировке.

Создание текстового файла:

```
if (GetOpenFileName(&ofn) == TRUE)
{
    //Открываем файл для чтения
    hFile = CreateFile(ofn.lpstrFile, GENERIC_READ,
FILE_SHARE_READ, NULL, OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL, NULL);
}
```

Чтение текстового файла:

```
ReadFile(hFile, str1, length_file, &length, NULL);
```

Запись в текстовый файл:

```
hFile = CreateFile(ofn.lpstrFile, GENERIC_WRITE,
FILE_SHARE_WRITE, NULL, CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL, NULL);
length = GetWindowTextLength(hEdit);
WriteFile(hFile, buf, length, &lng, NULL);
```

## ***Бинарные файлы***

Бинарные файлы содержат массивы данных чаще всего нетекстового вида, это могут быть как целые, так и вещественные числа, пользовательские структуры данных и т. п.

Создание файла:

```
hFile = CreateFile(str,           // имя файла
GENERIC_READ | GENERIC_WRITE,   // open for writing
0,                               // do not share
0,                               // default security
CREATE_ALWAYS,                  // overwrite existing
0,                               // normal file
0);                             // no attr. Template
```



Вариант создания двоичного файла с использованием стандартного диалога:

```
if (GetSaveFileName(&ofn)==true)
{
    hFile = CreateFile(ofn.lpstrFile,GENERIC_WRITE,
FILE_SHARE_WRITE, NULL, CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL, NULL);
}
```

### ***Запись данных в файл***

Арифметические данные в памяти (оперативная память, файлы) располагаются младшими байтами вперед, т. е. при выводе его дампа их наблюдают в «перевернутом» виде.

```
if (bIsBin)
{
    WriteFile(hFile, &svalue, sizeof(short), &cc, 0);
}
else // если текстовый, форматируем строку
    // и пишем в файл
{
    int i = 13;
    swprintf_s(str, 500, L"%d", svalue);
    WriteFile(hFile, &str, wcslen(str)*2 , &cc, 0);
}
```

### ***Чтение данных из двоичного файла в шестнадцатеричной кодировке***

```
// определяем длину файла
iLength = GetFileSize(hFile, NULL);
// устанавливаем указатель файла на начало файла
SetFilePointer(hFile, 0, NULL, FILE_BEGIN);
// считываем из него данные
ReadFile(hFile, sread, iLength, &iRead, NULL);
i = 0;
// форматируем строку и выводим в шестнадцатеричном виде
while (i < iLength)
{
    k += sprintf(q + k, "%x ", (unsigned
char)sread[i]);
    i++;
}
SetDlgItemTextA(hwnd, IE_DUMP, q);
```

Замечание. Данные различного типа занимают указанное количество байт:

int	// 4 байта
BYTE	// 1 байт
short	// 2 байта
long	// 4 байта
float	// 4 байта
double	// 8 байт
wchar_t	// 2 байт

### Дамп содержимого файла

На рис. 4.1 изображен дамп файла binary.bin, данные в нем имеют типы (в порядке их расположения в файле): short, byte, int, long, float, double, wchar\_t, wchar\_t.

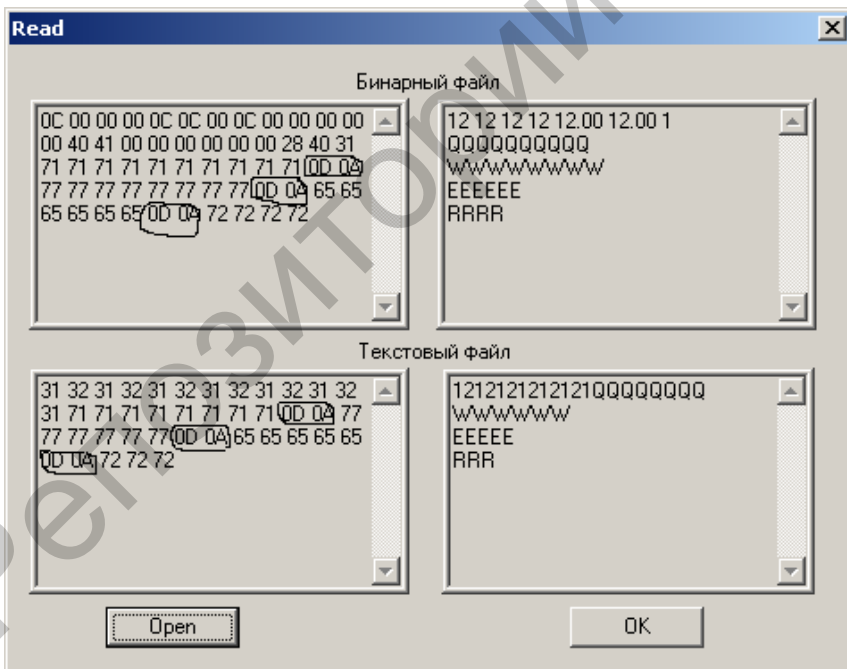


Рис. 4.1. Реальное содержимое и дампы бинарного и текстового файлов

## *Работа с каталогами и манипулирование файлами*

Прежде чем рассматривать функции API Win32, относящиеся к файловому вводу-выводу, отметим, как можно выяснить причину их ошибочного завершения. Для этого Windows предоставляет функцию *GetLastError*. Для вызова функции *GetLastError* не нужно передавать никаких параметров. Эту функцию необходимо вызвать сразу после функции API Win32/64 файлового ввода-вывода, успешность работы которой проверяется.

### *Поиск файлов*

Один из методов поиска файлов – просмотр содержимого всего диска. Начальный каталог и файл, который нужно найти, указываются системе через функцию *FindFirstFile*:

```
HANDLE FindFirstFile(LPTSTR IpszSearchFile,  
LPWIN32_FIND_DATA Ipffd).
```

Параметр *IpszSearchFile* указывает на строку (с нулевым символом в конце), содержащую имя файла. В имя можно включать символы подстановки (\* и ?) и начальный путь. Параметр *Ipffd* – адрес структуры *LPWIN32\_FIND\_DATA*:

```
typedef struct _WIN32_FIND_DATA{DWORD dwFileAttributes;  
    FILETIME ftCreationTime;  
    FILETIME ftLastAccessTime;  
    FILETIME ftLastWriteTime;  
    DWORD nFileSizeHigh;  
    DWORD nFileSizeLow;  
    DWORD dwReserved0;  
    DWORD dwReserved1;  
    CHAR cFileName[ MAX_PATH];  
    CHAR cAlternateFileName[14];  
}WIN32_FIND_DATA,*PWIN32_FIND_DATA,  
*LPWIN32_FIND_DATA;
```

Обнаружив файл, соответствующий заданной спецификации и расположенный в заданном каталоге, функция заполняет элементы структуры и возвращает описатель, а при отсутствии такого файла – возвращает код *INVALID\_HANDLE\_VALUE* и не изменяет содержимое структуры. Функция *FindFirstFile* в случае неудачи возвращает *INVALID\_HANDLE\_VALUE*, а не *NULL*.

Структура WIN32\_FIND\_DATA содержит информацию о найденном файле: атрибуты, метки времени и размер. Элемент с FileName содержит истинное имя файла. Его обычно и используют. В элемент с AlternateFileName помещается так называемый псевдоним файла.

Если FindFirstFile нашла подходящий файл, можно вызвать FindNextFile для поиска следующего файла, удовлетворяющего спецификации, переданной при вызове FindFirstFile:

```
BOOL FindNextFile(HANDLE hFindFile, LPWIN32_FIND_DATA  
lpfffd);
```

Первый ее параметр – описатель, полученный предыдущим вызовом FindFirstFile. Второй вновь указывает на структуру WIN32\_FIND\_DATA. Это может быть и другой экземпляр структуры, а не тот, что был указан при вызове FindFirstFile.

При успешном выполнении FindNextFile возвращает TRUE и заполняет структуру WIN32\_FIND\_DATA, в ином случае она возвращает FALSE.

Закончив поиск, надо закрыть описатель, возвращенный FindFirstFile, вызвав функцию

```
BOOL FindClose(HANDLE hFindFile).
```

FindFirstFile и FindNextFile позволяют просматривать только те файлы (и подкаталоги), что находятся в одном, указанном пользователем каталоге. Чтобы «пройти» по всей иерархии каталогов, придется написать свою рекурсивную функцию.

### ***Размер файла***

Размер файла получают вызовом GetFileSize:

```
DWORD GetFileSize(HANDLE hFile, LPDWORD lpdwFileSizeHigh),
```

где параметр hFile - описатель открытого файла.

GetFileSize возвращает младшие 32 бита числа, представляющего размер файла. Чтобы получить старшие 32 бита, следует передать ей в lpdwFileSizeHigh адрес переменной типа DWORD.

Размер файла можно изменить функцией SetEndOfFile.

## Создание и удаление каталогов

Представленные ниже три функции, как и подсказывают их имена, позволяют процессу создавать и удалять каталоги:

```
BOOL CreateDirectory(LPTSTR IpszPath, ATTRIBUTES Ipsa);  
BOOL CreateDirectoryEx(LPCTSTR IpTemplateDirectory,  
LPCTSTR IpNewDirectory,  
LPSECURITY_ATTRIBUTES IpSecurityAttributes);  
BOOL RemoveDirectory(LPTSTR IpszDir);
```

При создании каталога процесс может инициализировать структуру SECURITY\_ATTRIBUTES и присвоить каталогу особые атрибуты, чтобы, например, другой пользователь не мог «войти» в созданный каталог или удалить его. Разграничение доступа возможно только при создании каталога в разделе NTFS. Поскольку CreateDirectory и CreateDirectoryEx не возвращают описателей, элемент blnheritHandle структуры SECURITY\_ATTRIBUTES игнорируется.

CreateDirectoryEx позволяет присвоить новому каталогу атрибуты, связанные с каким-либо существующим каталогом (указав, например, надо ли сжимать файлы в каталоге).

При успешном выполнении все три функции возвращают TRUE, в ином случае – FALSE. RemoveDirectory дает ошибку, если каталог не пуст (т. е. содержит какие-то файлы или подкаталоги) или если процесс не имеет права на удаление этого каталога.

Перечислим некоторые функции API Win32, имеющие отношение к работе с файловой системой.

Для определения конфигурации системы используются следующие функции:

```
DWORD GetLogicalDriveStrings(  
    DWORD nBufferLength, // size of buffer  
    LPCTSTR lpBuffer      // drive strings buffer  
);
```

При успешном завершении функция возвращает количество символов, скопированных в буфер:

```
DWORD GetLogicalDrives(VOID);
```

Двойное слово, которое возвращает эта функция, фактически является логической шкалой, нулевой бит в которой соответствует диску А, первый бит – В и т. д.

Для более подробной информации о дисках Windows предоставляет следующую функцию:

```

BOOL GetVolumeInformation(
    LPCTSTR      lpRootPathName,    // root directory
    LPTSTR       lpVolumeNameBuffer, // volume name buffer
    DWORD        nVolumeNameSize,   // length of name buffer
    LPDWORD      lpVolumeSerialNumber, // volume serial number
    LPDWORD      lpMaximumComponentLength, // maximum file name
                                                //length
    LPDWORD      lpFileSystemFlags,  // file system options
    LPTSTR       lpFileSystemNameBuffer, // file system name
                                                //buffer
    DWORD        nFileSystemNameSize) // length of file
                                        //system name buffer
);

```

Для получения информации о разбивке тома, а также о свободном пространстве на диске предусмотрена функция

```

BOOL GetDiskFreeSpace(
    LPCTSTR      lpRootPathName,    // root path
    LPDWORD      lpSectorsPerCluster, // sectors per cluster
    LPDWORD      lpBytesPerSector,    // bytes per sector
    LPDWORD      lpNumberOfFreeClusters, // free clusters
    LPDWORD      lpTotalNumberOfClusters // total clusters);

```

Для получения полной информации о файле используется функция

```

BOOL GetFileInformationByHandle(
    HANDLE hFile, // handle to file
    LPBY_HANDLE_FILE_INFORMATION lpFileInformation // buffer
);

```

Второй аргумент этой функции является указателем на структуру типа `BY_HANDLE_FILE_INFORMATION`, которая заполняется этой функцией. Структура описана в файле `winbase.h`. Это описание выглядит следующим образом:

```

typedef struct _BY_HANDLE_FILE_INFORMATION{
    DWORD dwFileAttributes; //Атрибуты файла

```

```

FILETIME  ffCreationTime;      //Время создания файла
FILETIME  ffLastAccessTime;    //Время последнего доступа к файлу
FILETIME  ffLastWriteTime;     //Время последней записи в файл
DWORD     dwVolumeSerialNumber; //Серийный номер тома
DWORD     nFileSizeHigh;       // Старшие 32 разряда размера файла
DWORD     nFileSizeLow;        // младшие 32 разряда размера файла
DWORD     nNumberOfLinks       // число ссылок на файл
DWORD     nFileIndexHigh      // старшие 32 разряда идентификатора файла
DWORD     nFileIndexLow       // младшие 32 разряда идентификатора файла
} BY_HANDLE_FILE_INFORMATION,
*PBY_HANDLE_FILE_INFORMATION,
*LBY_HANDLE_FILE_INFORMATION

```

Для получения отдельной информации о файле служат следующие функции, по названию которых можно определить, какую информацию они возвращают:

```

DWORD GetFileAttributes(
LPCTSTR lpFilename      // name of file or directory);
DWORD GetFileSize( HANDLE hFile, // handle to file
LPDWORD lpFileSizeHigh // high-order word of file size
);
BOOL GetFileTime(HANDLE hFile,
LPFILETIME lpCreationTime, // creation time
LPFILETIME lpLastAccessTime, // last access time
LPFILETIME lpLastWriteTime // last write time
);

```

## **Лабораторная работа № 5**

### **ФАЙЛОВАЯ СИСТЕМА FAT**

**Цель работы:** ознакомление со структурой и расположением системной информации и файлов на томе FATxx.

#### **Изучаемые вопросы**

1. Структура и расположение системной информации на томе FATxx.
2. Загрузочная запись BOOT: структура, расположение на томе.
3. Таблица расположения файлов FAT: структура, расположение на томе.
4. Корневой директориий ROOT: структура, расположение на томе.
5. Директориий/каталог: структура, расположение, структура записи.
6. Изучить алгоритм расположения файла на FAT томе:
  - а) определение номера начального кластера файла.
  - б) расчет номеров цепочки кластеров файла.
7. Исследовать изменение элементов таблиц FAT, родительского каталога при выполнении над файлом команд COPY, MOVE, DEL, RENAME.
8. Атрибуты файла: кодировка, управление.
9. Программно определить дату и время создания файла с именем NAME, подчиненного корневому каталогу ROOT.

#### **Последовательность выполнения работы**

1. Создать на внешнем носителе (flash) логический диск и отформатировать его под FAT16 или FAT32. Все дальнейшие действия выполняются с этим диском/томом.
2. Загрузить программу WinHex и скопировать с этого диска первый сектор системных данных – BOOT, FAT и ROOT в файлы BOOT.bin, FAT.bin, ROOT.bin.
3. Создать программу, которая:
  - а) выводит на экран дамп  $N$ -го сектора;
  - б) выводит на экран расшифрованную таблицу BPB;
  - в) определяет положение и размер системных данных BOOT, FAT, ROOT – и выводит их значения на экран.



4. Сравнить данные, полученные разработанной программой и WinHex. В случае несовпадения откорректировать алгоритм и отладить программу.

5. Создать на томе каталог и скопировать 1-й сектор каталога в файл DIR.bin.

6. Использовать дампы (фрагменты дампов) полученных файлов и фрагменты копий окон программ для подготовки отчета об изучаемых вопросах 1–5.

7. Изучить алгоритм расположения файла на FAT томе:

- а) определение номера начального кластера файла;
- б) расчет номеров цепочки кластеров файла. Данный пункт выполняется с помощью программы WinHex и не требует написания кода. В отчете привести алгоритм и иллюстрировать ответ фрагментами дампов родительского директория и таблицы FAT.

8. С помощью WinHex получить дампы таблицы FAT и родительского каталога до и после выполнения каждой команды COPY, MOVE, DEL и RENAME любым программным средством. Исследовать изменение элементов FAT и каталога.

### Теоретические и справочные сведения

Самый первый сектор любого жесткого диска имеет название Master Boot Record (**MBR**). Он состоит из двух частей: программы начального загрузчика и таблицы разделов (Disk Partition Table, **DPT**). Структура **MBR** приведена в табл. 5.1.

Таблица 5.1

Структура MBR

Смещение	Описание	Значение
0000h-01BDh	Код начальной загрузки	<b>MBR</b>
01BEh-01CDh	Описатель 1-го основного раздела	<b>Partition Table</b>
01CEh-01DDh	Описатель 2-го основного раздела	
01DEh-01EDh	Описатель 3-го основного раздела	
01EEh-01FDh	Описатель 4-го основного раздела	
01FEh-01FFh	Сигнатура системного байта (0xAA55)	—

DPT состоит из четырех записей – описателей раздела, обозначающих адрес начала раздела, его размер в секторах, адрес конца и тип файловой системы. Часто используются только две записи – для основного и расширенного раздела, потому что каждый подраздел (логический диск) имеет такую же DPT с указанием адреса следующего подраздела. Структура записи раздела приведена в табл. 5.2.

Таблица 5.2

Описатель раздела

Смещение	Описание	Примечание
0000h	Маркер начальной загрузки	–
0001h	Головка	Начало размещения
0002h	Сектор и цилиндр (биты 8–9)	
0003h	Цилиндр (биты 0–7)	
0004h	Системное описание	–
0005h	Головка	Конец размещения
0006h	Сектор и цилиндр (биты 8–9)	
0007h	Цилиндр (биты 0–7)	
0008h-000Bh	Смещение секторов	–
000Ch-000Fh	Количество секторов в разделе	–

Самый первый сектор раздела называется **Boot Record**. В его составе также есть загрузочная программа и таблица **BIOS Parameter Block (BPB)** (табл. 5.3–5.5). В этой таблице содержится информация о местонахождении и размере другой важной области – таблицы размещения файлов – FAT. По своей сути FAT – это большой массив элементов, каждый элемент соответствует определенному кластеру тома (табл. 5.6).

Задача загрузчика в MBR – дать возможность загрузки модулей (файлов) операционной системы с нужного раздела (т. е. использовать несколько операционных систем). Загрузка возможна с того раздела, у которого установлен флаг активности таблице DPT.

Таблица 5.3

## Формат загрузочной записи

Смещение (байт)	Размер (байт)	Содержание
0 (00h)	3	Команда JMP xxxxx – ближний переход на программу начальной загрузки
3 (03h)	8	Название фирмы-изготовителя ОС и версия
11 (0Bh)	25	Расширенный блок параметров BIOS
36 (24h)	1	Физический номер устройства (0 – НГМД, 80h – НМД)
37 (25h)	1	Зарезервировано
38 (26h)	1	29h – признак расширенной загрузочной записи
39 (27h)	4	Серийный номер диска, создается во время форматирования
43 (2Bh)	11	Метка диска
54 (36h)	8	Содержит запись 'FAT12 ' или 'FAT16 ', которая идентифицирует формат FAT
		Начальный загрузчик операционной системы
510 (1FE)	2	Сигнатура 55h AAh

Таблица 5.4

## Блок параметров BIOS для FAT16

Смещение (байт)	Размер (байт)	Содержание
0 (00h)	2	Количество байт в одном секторе диска
2 (02h)	1	Количество секторов в одном кластере
3 (03h)	2	Количество зарезервированных секторов
5 (05h)	1	Количество FAT
6 (06h)	2	Максимальное количество дескрипторов файлов в корневом каталоге диска
8 (08h)	2	Общее количество секторов на носителе данных
10 (0Ah)	1	Байт-описатель среды носителя данных (media)
11 (0Bh)	2	Количество секторов, занимаемых одной копией FAT
13 (0Dh)	2	Количество секторов на дорожке

Смещение (байт)	Размер (байт)	Содержание
15 (0Fh)	2	Количество магнитных головок
17 (11h)	2	Количество скрытых секторов для носителя размером до 32 Мб
19 (13h)	2	Количество скрытых секторов для носителя размером свыше 32 Мб
21 (15h)	4	Общее количество секторов на логическом диске, превышающим по размеру 32 Мб

Таблица 5.5

## Boot Record для FAT32

Смещение	Размер, байт	Описание
0x00	3	Безусловный переход (jmp) на загрузочный код
0x03	8	Идентификатор фирмы-изготовителя
0x0B	2	Число байт в секторе
0x0D	1	Число секторов в кластере
0x0E	2	Число резервных секторов в резервной области раздела, начиная с первого сектора раздела
0x10	1	Число таблиц FAT
0x11	2	Для FAT12/FAT16 – количество 32-байтных дескрипторов файлов в корневом каталоге, для FAT32 – 0
0x13	2	Общее число секторов в разделе; если данное поле содержит ноль, то число секторов задается полем со смещением 0x20
0x15	1	Тип носителя. Для жесткого диска имеет значение 0xF8
0x16	2	Для FAT16 – количество секторов, занимаемых одной копией FAT. Для FAT32 имеет значение 0
0x18	2	Число секторов на дорожке (для прерывания 0x13)
0x1A	2	Число рабочих поверхностей (для прерывания 0x13)
0x1C	4	Число скрытых секторов перед разделом

Смещение	Размер, байт	Описание
0x20	4	Общее число секторов в разделе. Поле используется, если в разделе свыше 65535 секторов, в противном случае поле содержит 0
0x24	1	Номер дисковода для прерывания 0x13
0x25	1	Зарезервировано для Windows NT, имеет значение 0
0x26	1	Признак расширенной загрузочной записи (0x29)
0x27	4	Номер логического диска
0x2B	11	Метка диска
0x36	8	Текстовая строка с аббревиатурой типа файловой системы

Загрузочный сектор логического диска должен содержать в байте со смещением 0x1FE код 0x55, а в следующем байте (смещение 0x1FF) – код 0xAA. Указанные два байта являются признаком загрузочного диска. Таким образом, загрузочный сектор выполняет две важные функции: описывает структуру данных на диске, а также позволяет осуществить загрузку операционной системы.

Размер сектора в 512 байт

Boot Record disk C:

```

00000000: EB 58 90 4D 53 57 49 4E - 34 2E 31 00 02 10 20 00 .X.MSWIN4.1...
00000010: 02 00 00 00 00 F8 00 00 - 3F 00 FF 00 3F 00 00 00 .....?..?...
00000020: 51 50 1E 01 C6 23 00 00 - 00 00 00 00 02 00 00 00 QP...#.....
00000030: 01 00 06 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....
00000040: 80 00 29 ED 1E 19 32 53 - 59 53 54 45 4D 5F 31 20 ...}...2SYSTEM 1
00000050: 20 20 46 41 54 33 32 20 - 20 20 33 C9 8E D1 BC F4 FAT32 3....
00000060: 7B 8E C1 8E D9 BD 00 7C - 88 4E 02 8A 56 40 B4 08 {...|.N.V@..
00000070: CD 13 73 05 B9 FF FF 8A - F1 66 0F B6 C6 40 66 0F ...s.....f...@f.
00000080: B6 D1 80 E2 3F F7 E2 86 - CD C0 ED 06 41 66 0F B7 .....?.....Af..
00000090: C9 66 F7 E1 66 89 46 F8 - 83 7E 16 00 75 38 83 7E .f..f.F...~.u8..
000000A0: 2A 00 77 32 66 8B 46 1C - 66 83 C0 0C BB 00 80 B9 *.w2f.F.f.....
000000B0: 01 00 E8 2B 00 E9 48 03 - A0 FA 7D B4 7D 8B F0 AC ...+.H...}.)...
000000C0: 84 C0 74 17 3C FF 74 09 - B4 0E BB 07 00 CD 10 EB .t.<.t.....
000000D0: EE A0 FB 7D EB E5 A0 F9 - 7D EB E0 98 CD 16 CD 19 ...).....
000000E0: 66 60 66 3B 46 F8 0F 82 - 4A 00 66 6A 00 66 50 06 f'f;F...J.fj.fP.
000000F0: 53 66 68 10 00 01 00 80 - 7E 02 00 0F 85 20 00 B4 Sfh.....~....

```

```

00000100: 41 BB AA 55 8A 56 40 CD - 13 0F 82 1C 00 81 FB 55 A..U.V@.....U
00000110: AA 0F 85 14 00 F6 C1 01 - 0F 84 0D 00 FE 46 02 B4 .....F..
00000120: 42 8A 56 40 8B F4 CD 13 - B0 F9 66 58 66 58 66 58 B.V@.....fXfXfX
00000130: 66 58 EB 2A 66 33 D2 66 - 0F B7 4E 18 66 F7 F1 FE fX.*f3.f..N.f...
00000140: C2 8A CA 66 8B D0 66 C1 - EA 10 F7 76 1A 86 D6 8A ...f..f...v....
00000150: 56 40 8A E8 C0 E4 06 0A - CC B8 01 02 CD 13 66 61 V@.....f@I..q...
00000160: 0F 82 54 FF 81 C3 00 02 - 66 40 49 0F 85 71 FF C3 ..T.....f@I..q...
00000170: 4E 54 4C 44 52 20 20 20 - 20 20 20 00 00 00 00 00 NTLDR .....
00000180: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....
00000190: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 .....
000001A0: 00 00 00 00 00 00 00 00 - 00 00 00 00 0D 0A 4E 54 .....NT
000001B0: 4C 44 52 20 69 73 20 6D - 69 73 73 69 6E 67 FF 0D LDR is missing..
000001C0: 0A 44 69 73 6B 20 65 72 - 72 6F 72 FF 0D 0A 50 72 .Disk error...Pr
000001D0: 65 73 73 20 61 6E 79 20 - 6B 65 79 20 74 6F 20 72 ess any key to r
000001E0: 65 73 74 61 72 74 0D 0A - 00 00 00 00 00 00 00 00 estart.....
000001F0: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 55 AA .....U.

```

Для **FAT32** Boot Record занимает 6 секторов.

Размер сектора в 512 байт

Таблица 5.6

### Значения элементов в FAT

FAT 12	FAT 16	Тип кластера
000h	0000h	Свободный кластер
FF0h-FF6h	FFF0h-FFF6h	Зарезервированный кластер
FF7h	FFF7h	Плохой кластер
FF8h-FFFh	FFF8h-FFFFh	Последний кластер в списке
002h – FEFh	0002h-FFEFh	Номер следующего кластера в списке

Формат дескриптора каталога, байт атрибутов файла, структура записи с длинным именем, префиксы в именах каталогов, формат поля времени и даты обновления файла представлены в табл. 5.7–5.11.

Таблица 5.7

### Формат дескриптора каталога (short name)

Смещение, байт	Размер, байт	Содержание
0 (00h)	8	Имя файла или каталога, выровненное на левую границу и дополненное пробелами
8 (08h)	3	Расширение имени файла, выровненное на левую границу и дополненное пробелами
11 (0Bh)	1	Атрибуты файла
12 (0Ch)	10	Зарезервировано

Смещение (байт)	Размер, байт	Содержание
22 (16h)	2	Время создания файла или время его последней модификации
24 (18h)	2	Дата создания файла или дата его последней модификации
26 (1Ah)	2	Номер первого кластера, распределенного файлу
28 (1Ch)	4	Размер файла в байтах

Таблица 5.8

## Байт атрибутов файла (0Bh в дескрипторе)

Биты атрибута	Значение
0	Файл предназначен только для чтения, в этот файл нельзя писать и его нельзя стирать
1	Скрытый файл, этот файл не будет появляться в списке файлов, создаваемом командой операционной системы
2	Системный файл. Этот бит обычно установлен в файлах, являющихся составной частью операционной системы
3	Данный дескриптор описывает метку диска. Для этого дескриптора поля имени файла и расширения имени файла должны рассматриваться как одно поле длиной 11 байтов. Это поле содержит метку диска
4	Дескриптор описывает файл, являющийся подкаталогом данного каталога
5	Флаг архивации
6	Зарезервирован
7	Зарезервирован

Таблица 5.9

## Структура записи с длинным именем

Смещение, байт	Размер, байт	Содержание
0 (00h)	1	Номер записи, принадлежащей одному из файлов или каталогу. Последняя запись дополняется маской 40h (максимальное количество записей)

Окончание табл. 5.9

Смещение, байт	Размер, байт	Содержание
1 (01h)	10	5 последующих символов из длинного имени файла или каталога в 2-байтовой кодировке
11 (0Bh)	1	Атрибут 0Fh
12 (0Ch)	1	Зарезервировано и должно быть равным нулю
13 (0Dh)	1	Контрольная сумма короткого имени
14 (0Eh)	12	Следующие 6 символов составного имени после предыдущих 5
26 (1Ah)	2	Должно быть равным нулю
28 (1Ch)	4	Следующие 2 символа составного имени после предыдущих 6

Таблица 5.10

#### Префиксы в именах каталогов (файлов)

Байт	Значение
E5h	Директория удалена
00h	Директория не используется (не размечена)
05h	Используется японский шрифт

Таблица 5.11

#### Формат поля времени

Биты формата	Значение
15–11	Часы (0–23)
10–5	Минуты (0–59)
4–0	Секунды/2 (0–29)

Таблица 5.12

#### Формат даты обновления файла (напоминает формат времени)

Биты формата	Значение
15–9	Год (0–119) + 1980
8–5	Месяц (1–12)
4–0	День (1–31)



## *Лабораторная работа № 6*

### **ФАЙЛОВАЯ СИСТЕМА NTFS**

**Цель работы:** ознакомление со структурой и расположением системной информации и данных на томе NTFS.

#### **Изучаемые вопросы**

1. Структура BOOT.
2. Таблица файлов MFT.
3. Записи главной таблицы файлов (FILE RECORD).
4. Метафайлы.
5. Структура файла (небольшого и большого).
6. Атрибуты файла.
7. Каталоги в NTFS (структура).
8. Изучить алгоритм поиска расположения файлов на диске:
  - а) определение номера начального кластера расположения файла на диске;
  - б) расчет номеров кластеров файла на диске;
9. Исследовать изменение элементов системных файлов NTFS при выполнении команд COPY, MOVE, DEL, RENAME.

#### **Постановка задачи**

Разработать программу, которая выводит на экран:

1. Дамп 1-го сектора BOOT и расшифровывает структуру METADATABOOT.
2. Список атрибутов короткого файла и тело атрибута \$DATA. Файл создать любым текстовым редактором в кодировке ANSI, его содержание – две строки: первая – ФИО студента, вторая – дата его рождения.

В отчете привести дампы и расшифровать структуры метаданных (1-й сектор): BOOT, файл \$MFT (индекс записи соответствует номеру варианта), записи файла \$MFT, которые соответствуют короткому и длинному файлам, любой директорий (один сектор).

## Теоретические сведения

### Структура Boot

Загрузочная запись тома под NTFS (BOOT), табл. 6.1, содержит основную информацию о томе (логическом диске), такую как расположение MFT, количество секторов на кластер, всего секторов на томе, код загрузчика (NT Loader) и т. д.

Таблица 6.1

### Структура BOOT

Offset	Size	Description	typedef struct fileBoot
0x0000	3	Jump to the boot loader routine	{
0x0003	8	System Id: "NTFS "	BYTE dJump[3];
0x000B	2	Bytes per sector	BYTE dSystemId[8];
0x000D	1	Sectors per cluster	WORD dBytesPerSector;
0x000E	7	Unused	BYTE dSectorPerCluster;
0x0015	1	Media descriptor (a)	BYTE dUnusedA[7];
0x0016	2	Unused	BYTE dMediaId;
0x0018	2	Sectors per track	BYTE dUnusedB[2];
0x001A	2	Number of heads	WORD dSectorPerTrack;
0x001C	8	Unused	WORD dHeadsCount;
0x0024	4	Usually 80 00 80 00 (b)	BYTE dUnusedC[8];
0x0028	8	Number of sectors in the volume	BYTE dUsually[4];
0x0030	8	LCN of VCN 0 of the \$MFT	INT64 dNumberOfSectors;
0x0038	8	LCN of VCN 0 of the \$MFTMirr	INT64 dLCNofMFT;
0x0040	4	Clusters per MFT Record (c)	INT64 dLCNofMFTMirr;
0x0044	4	Clusters per Index Record (c)	DWORD ClusterPerMFT;
0x0048	8	Volume serial number	DWORD ClusterPerIndexes
~	~	~	BYTE dSerialNumber[8];
0x0200		Windows NT Loader	BYTE dDataCode[432];
			} METADATABOOT;

### Таблица файлов MFT

Каждый файл на томе NTFS представлен записью в специальном файле, называемом главной файловой таблицей (MFA – master file table). NTFS резервирует первые 16 записей таблицы для специальной информации. Первая запись этой таблицы описывает непосредственно главную файловую таблицу. За ней следует зеркальная запись (mirror record) MFT. Если первая запись MFT разрушена, то

NTFS читает вторую запись для отыскания зеркального файла MFT, первая запись которого идентична первой записи MFT. Местоположения сегментов данных MFT и зеркального файла MFT записаны в секторе начальной загрузки. Дубликат сектора начальной загрузки находится в логическом центре диска.

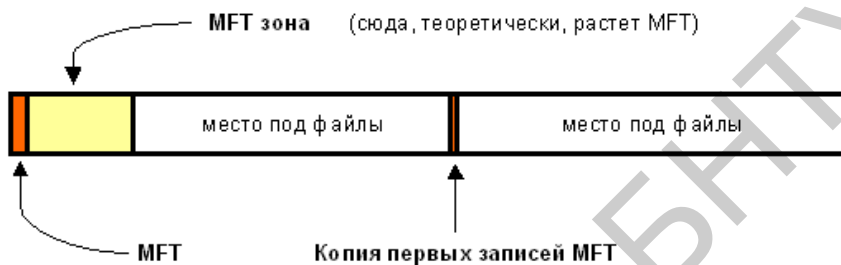


Рис. 6.1. Расположение MFT на диске

### ***Метафайлы***

Первые 16 файлов NTFS (метафайлы) носят служебный характер. Каждый из них отвечает за какой-либо аспект работы системы. Метафайлы находятся в корневом каталоге NTFS диска – их имена начинаются с символа "\$" (табл. 6.2).

Таблица 6.2

## Метафайлы NTFS

Inode	Filename	OS	Description
0	\$MFT		Master File Table - An index of every file
1	\$MFTMirr		A backup copy of the first 4 records of the MFT
2	\$LogFile		Transactional logging file
3	\$Volume		Serial number, creation time, dirty flag
4	\$AttrDef		Attribute definitions
5	.(dot)		Root directory of the disk
6	\$Bitmap		Contains volume's cluster map (in-use vs. free)
7	\$Boot		Boot record of the volume
8	\$BadClus		Lists bad clusters on the volume
9	\$Quota	NT	Quota information
9	\$Secure	2K	Security descriptors used by the volume
10	\$UpCase		Table of uppercase characters used for collating
11	\$Extend	2K	A directory: \$ObjId, \$Quota, \$Reparse, \$UsnJrnl
12-15	<Unused>		Marked as in use but empty
16-23	<Unused>		Marked as unused
Any	\$ObjId	2K	Unique Ids given to every file
Any	\$Quota	2K	Quota information
Any	\$Reparse	2K	Reparse point information
Any	\$UsnJrnl	2K	Journaling of Encryption
>24	A_File		An ordinary file
>24	A_Dir		An ordinary directory
...	...		...

Каждая запись в MFT начинается с заголовка (табл. 6.3), за которым следует набор атрибутов (табл. 6.4).

Таблица 6.3

## Структура заголовка записи в MFT

Offset	Size	OS	Description
0x00	4		Magic number 'FILE'
0x04	2		Offset to the update sequence
0x06	2		Size in words of Update Sequence Number & Array (S)
0x08	8		\$LogFile Sequence Number (LSN)
0x10	2		Sequence number
0x12	2		Hard link count
0x14	2		Offset to the first Attribute
0x16	2		Flags
0x18	4		Real size of the FILE record
0x1C	4		Allocated size of the FILE record
0x20	8		File reference to the base FILE record
0x28	2		Next Attribute Id
0x2A	2	XP	Align to 4 byte boundary
0x2C	4	XP	Number of this MFT Record
	2		Update Sequence Number (a)
	2S-2		Update Sequence Array (a)

Таблица 6.4

## Стандартные атрибуты NTFS

Type	OS	Name
0x10		\$STANDARD_INFORMATION
0x20		\$ATTRIBUTE_LIST
0x30		\$FILE_NAME
0x40	NT	\$VOLUME_VERSION
0x40	2K	\$OBJECT_ID
0x50		\$SECURITY_DESCRIPTOR
0x60		\$VOLUME_NAME
0x70		\$VOLUME_INFORMATION
0x80		\$DATA
0x90		\$INDEX_ROOT
0xA0		\$INDEX_ALLOCATION
0xB0		\$BITMAP
0xC0	NT	\$SYMBOLIC_LINK
0xC0	2K	\$REPARSE_POINT
0xD0		\$EA_INFORMATION
0xE0		\$EA
0xF0	NT	\$PROPERTY_SET
0x100	2K	\$LOGGED_UTILITY_STREAM

*Атрибуты файла*

NTFS просматривает каждый файл (или каталог) как набор атрибутов файла. Такие элементы, как имя файла, информация защиты и даже данные – все это атрибуты файла. Каждый атрибут идентифицирован кодом типа атрибута и необязательно именем атрибута. Если атрибут достаточно велик, то он помещается в отдельном файле (нерезидентный атрибут). Если данных в файле не много, то они хранятся в записи файла MFT (рис. 6.2).



Рис. 6.2. Структура записи в MFT

### Записи главной таблицы файлов

Файл в NTFS есть ни что иное, как набор атрибутов (см. рис. 6.2). Атрибут представляется в виде потока байтов. Как видим, один из атрибутов – это данные, хранящиеся в файле, или, как говорят, – поток данных. Файловая система допускает добавление файлу новых атрибутов, которые могут содержать какие-то дополнительные данные.

Каждый атрибут имеет заголовок, структура которого зависит от того, резидентный атрибут или нет, а также имеет ли он имя. Каждый заголовок атрибута, как и сам атрибут, имеет определенную структуру (см. табл. 6.5).

Таблица 6.5

#### Структура заголовка атрибута (резидентный, не именованный)

Offset	Size	Value	Description
0x00	4		Attribute Type (e.g. 0x10, 0x60)
0x04	4		Length (including this header)
0x08	1	0x00	Non-resident flag
0x09	1	0x00	Name length
0x0A	2	0x00	Offset to the Name
0x0C	2	0x00	Flags
0x0E	2		Attribute Id (a)
0x10	4	L	Length of the Attribute
0x14	2	0x18	Offset to the Attribute
0x16	1		Indexed flag
0x17	1	0x00	Padding
0x18	L		The Attribute

Если данные файла не помещаются в одну запись MFT, то этот факт отражается в заголовке атрибута (см. табл. 6.5) Data, который содержит признак того, что этот атрибут является нерезидентным, то есть находится в отрезках вне таблицы MFT. В этом случае атрибут Data содержит адресную информацию каждого отрезка данных (табл. 6.6 – 6.8).

Таблица 6.6

Структура заголовка атрибута  
(нерезидентный, неименованный)

Offset	Size	Value	Description
0x00	4		Attribute Type (e.g. 0x20, 0x80)
0x04	4		Length (including this header)
0x08	1	0x01	Non-resident flag
0x09	1	0x00	Name length
0x0A	2	0x00	Offset to the Name
0x0C	2		Flags
0x0E	2		Attribute Id (a)
0x10	8		Starting VCN
0x18	8		Last VCN
0x20	2	0x40	Offset to the Data Runs
0x22	2		Compression Unit Size (b)
0x24	4	0x00	Padding
0x28	8		Allocated size of the attribute (c)
0x30	8		Real size of the attribute
0x38	8		Initialized data size of the stream (d)
0x40			Data Runs



Таблица 6.7

## Структура атрибута \$FILE\_NAME

Offset	Size	Description
~	~	Standard Attribute Header
0x00	8	File reference to the parent directory.
0x08	8	C Time - File Creation
0x10	8	A Time - File Altered
0x18	8	M Time - MFT Changed
0x20	8	R Time - File Read
0x28	8	Allocated size of the file
0x30	8	Real size of the file
0x38	4	Flags, e.g. Directory, compressed, hidden
0x3c	4	Used by EAs and Reparse
0x40	1	Filename length in characters (L)
0x41	1	Filename namespace 0x42 2L File name in Unicode (not null terminated)

Таблица 6.8

## Структура атрибута \$STANDARD\_INFORMATION

Offset	Size	OS	Description
~	~		Standard Attribute Header
0x00	8		C Time - File Creation
0x08	8		A Time - File Altered
0x10	8		M Time - MFT Changed
0x18	8		R Time - File Read
0x20	4		DOS File Permissions
0x24	4		Maximum Number of Versions
0x28	4		Version Number
0x2C	4		Class Id
0x30	4	2K	Owner Id
0x34	4	2K	Security Id
0x38	8	2K	Quota Charged
0x40	8	2K	Update Sequence Number (USN)

## *Лабораторная работа № 7*

### **ПРОЦЕССЫ (часть 1)**

**Цель работы:** ознакомление с основами создания и управления процессами в ОС WINDOWS.

#### **Изучаемые вопросы**

1. Виды процессов.
2. Структуры STARTUPINFO, PROCESS\_INFORMATION.
3. Создание процесса.
4. Класс приоритета процесса.
5. Идентификатор процесса.
6. Дочерние процессы.
7. Наследование дочерними процессами ресурсов (файлов) родительского процесса.
8. Функции Win32 для управления процессами.
9. Окончание процесса.

#### **Постановка задачи**

1. Разработать приложение, состоящее из трех процессов (головного и двух дочерних). В головном процессе создаются/открываются два файла: текстовый (несколько строк) и двоичный (ряд арифметических данных). Информация об открытых файлах (дескрипторы) передается в дочерние процессы на этапе их создания через командную строку.

2. В главном процессе запускаются два дочерних процесса, которые:

- первый – дописывает строку/строки в текстовый файл;
- второй – дописывает число/числа в бинарный файл;
- процессы отображают информацию файлов до и после их модификации.

3. После возвращения из дочерних процессов главный процесс отображает:

- содержимое модифицированных файлов и закрывает их;
- системную информацию о процессе (GetProcessInformation).

## Теоретические сведения

### *Структуры STARTUPINFO, PROCESS\_INFORMATION*

При создании процесса как минимум должен быть задан размер структуры типа STARTUPINFO (поле **cb**) и обнулены остальные ее поля, например:

```
STARTUPINFO si;  
  
ZeroMemory( &si, sizeof(si) );  
si.cb = sizeof(si);  
ZeroMemory( &pi, sizeof(pi) );
```

PROCESS\_INFORMATION – структура, возвращаемая функцией CreateProcess. В ней определены такие поля, как дескрипторы созданного процесса и его главного потока, их идентификаторы. Например с помощью PROCESS\_INFORMATION находим дескриптор дочернего процесса:

PROCESS\_INFORMATION pi;

```
CreateProcess(NULL, str, NULL, NULL, true, NULL, NULL,  
,NULL, &si, &pi);  
SetPriorityClass(pi.hProcess, NORMAL_PRIORITY_CLASS);  
WaitForSingleObject(pi.hProcess, INFINITE);
```

### *Создание процесса*

Процесс создается с помощью функции CreateProcess(), которая возвращает TRUE в случае удачного создания процесса:

```
CreateProcess  
(NULL,           // имя исполняемого файла  
cmdLine,         // командная строка, передаваемая процессу  
                // определяет нужные атрибуты защиты  
                // для объектов "процесс" и "поток"  
                // соответственно, если NULL, то система закрепит за  
                // данными объектами дескрипторы защиты по умолчанию  
NULL,           //  
NULL,           //  
TRUE,           // система передаст процессу  
                // все наследуемые описатели  
0,              // флаги, влияющие на то, как именно  
                // создается новый процесс
```

```

NULL,      // дочерний процесс наследует строки
           // переменных окружения от родительского процесса
NULL,      // рабочий каталог нового процесса будет
           // тем же, что и у приложения, его породившего
&st1,      // указатель на структуру STARTUPINFO
&pi1,      // указатель на структуру PROCESS_INFO
);

```

В программе:

```

j = swprintf_s(str, 1000, L"%s",
L"D:\\УЧЕБА\\ishodnikiC++\\TextProcess\\debug\\TextProces
s\\");
swprintf_s(str + j, 1000 - j, L"%d", hFileText);

CreateProcess(NULL, str, NULL, NULL, true, NULL,
NULL, NULL, &si, &pi);

```

### ***Класс приоритета процесса***

Параметр `fdwCreate` функции `CreateProcess()` позволяет задать и класс приоритета процесса. Возможные классы приоритета перечислены в табл. 7.1.

Таблица 7.1

#### **Классы приоритетов процесса**

<b>Класс приоритета</b>	<b>Идентификатор</b>
Idle (простаивающий)	IDLE_PRIORITY_CLASS
Below normal (ниже обычного)	BELOW_NORMAL_PRIORITY_CLASS
Normal (обычный)	NORMAL_PRIORITY_CLASS
Above normal (выше обычного)	ABOVE_NORMAL_PRIORITY_CLASS
High (высокий)	HIGH_PRIORITY_CLASS
Realtime (реального времени)	REALTIME_PRIORITY_CLASS

При создании процесса можно задать класс его приоритета. По умолчанию система присваивает процессу класс приоритета `NORMAL_PRIORITY_CLASS`. Классы приоритета влияют на распределение процессорного времени между потоками процессов.

Задание приоритета на этапе создания процесса:

```
BOOL Res = CreateProcess(NULL, szRunAppl, NULL, NULL,
TRUE, NORMAL_PRIORITY_CLASS, NULL, NULL, &si, &pi);
```

Изменение класса приоритета процесса:

```
SetPriorityClass(prtInfo.hProcess,
ABOVE_NORMAL_PRIORITY_CLASS);
...
SetPriorityClass(prtInfo.hProcess, IDLE_PRIORITY_CLASS);
```

### ***Наследование дочерними процессами ресурсов родительского процесса***

Для наследования дочерними процессами ресурсов (файлов) родительского процесса в нем необходимо объявить структуру **sa** (**SECURITY\_ATTRIBUTES**). Третий параметр **sa** устанавливает права по совместному доступу к созданному файлу:

```
sa.nLength = sizeof(SECURITY_ATTRIBUTES);
sa.lpSecurityDescriptor = NULL;
sa.bInheritHandle = TRUE;
```

Затем в функции создания файла (родительский процесс) необходимо передать указатель на данную структуру:

```
hFileText = CreateFile(L"FileText.txt", GENERIC_READ
| GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE, &sa,
CREATE_ALWAYS, 0, 0);
```

Для использования файла в дочернем процессе необходимо передать его описатель **hFileText** через параметры командной строки. Например, в родительском процессе это можно реализовать следующим образом:

```
j = swprintf_s(str, 1000, L"%s",
L"D:\\УЧЕБА\\ishodnikiC++\\TextProcess\\debug\\TextProces
s\\");
swprintf_s(str + j, 1000 - j, L"%d", hFileText);
CreateProcess(NULL, str, NULL, NULL, TRUE, NULL, NULL,
NULL, &si, &pi);
```

В дочернем процессе описатель файла получают следующим образом:

```
LPTSTR str = GetCommandLine();
LPTSTR sp = StrRStrI(str, NULL, L"\"");
sp++;
HANDLE hFile = (HANDLE)StrToInt(sp);
```

В процедуре `CreateProcess` пятый параметр устанавливается в **TRUE**, что дает возможность созданному процессу наследовать все доступные (разрешено наследование) дескрипторы объектов ядра, принадлежащие родительскому процессу. Для этого необходимо разрешить наследование дескрипторов объектов ядра. Это делается с помощью структуры `SECURITY_ATTRIBUTES`, а именно ее флага `bInheritHandle`, например:

```
SECURITY_ATTRIBUTES sa;  
...  
sa.nLength = sizeof(SEcurity_ATTRIBUTES);  
sa.lpSecurityDescriptor = NULL;  
sa.bInheritHandle = TRUE;
```

### ***Функции Win32 для управления процессами***

Для управления процессами существует ряд API-функций:

`CreateProcess(NULL, str, NULL, NULL, true, NULL, NULL, NULL, &si, &pi)` – функция создает новый процесс;

`WaitForSingleObject(pi.hProcess, INFINITE)` – функция приостанавливает выполнение потока родительского процесса, пока не завершится дочерний процесс `pi.hProcess`;

`CloseHandle(hFileText)` – уменьшает значения счетчиков объектов на единицу. Когда счетчик уменьшится до нуля, освобождается память, занимаемая этим объектом;

`PTSTR GetCommandLine()` – получает указатель на полную командную строку;

`DWORD GetEnvironmentVariable(PCTSTR pszName, PTSTR pszValue, DWORD cchValue)` – позволяет выявлять присутствие той или иной переменной окружения и определять ее значение;

`DWORD GetCurrentDirectory(DWORD cchCurDir, PTSTR pszCurDir)`; – получает текущий каталог и диск для процесса;

`BOOL SetCurrentDirectory(PCTSTR pszCurDir)` – устанавливает текущий каталог и диск для процесса;

`HANDLE OpenProcess(DWORD fdwAccess, BOOL fInherit, DWORD, IDProcess)` – возвращает дескриптор существующего в системе процесса.

## *Окончание процесса*

Процесс можно завершить четырьмя способами:

- 1) входная функция первичного потока возвращает управление (рекомендуемый способ);
- 2) один из потоков процесса вызывает функцию `ExitProcess` (нежелательный способ);
- 3) поток другого процесса вызывает функцию `TerminateProcess` (тоже нежелательно);
- 4) все потоки процесса умирают по своей воле (большая редкость).

## **Лабораторная работа № 8**

### **ПРОЦЕССЫ (Часть 2)**

**Цель работы:** ознакомление с основами создания и управления процессами в ОС WINDOWS.

#### **Изучаемые вопросы**

1. Переменные окружения процесса: структура, значения.
2. Передача информации между процессами через среду процесса.
3. Список процессов.
4. Кто родитель процесса.
5. Текущая рабочая директория процесса.
6. Время выполнения процесса.
7. Наследование дочерними процессами среды родительского процесса.

#### **Постановка задачи**

1. Модифицировать программу лабораторной работы № 7 так, чтобы она выводила в окно информацию по изучаемым вопросам. Приложение может состоять из трех процессов (головного и двух дочерних). В родительском процессе создается и выводится список процессов, который обновляется через 3 с, и другая информация по процессу.

2. В главном процессе запускаются два дочерних процесса, где первый выводит PID, среду и PID родителя, а второй дописывает число в бинарный файл. Первый дочерний процесс записывает отображаемую информацию в текстовый файл.

3. После возвращения из дочерних процессов главный процесс отображает содержимое файла.

#### **Теоретические сведения**

##### ***Переменные окружения процесса***

С любым процессом связан блок переменных окружения (среда) – область памяти, выделенная в адресном пространстве процесса, ко-



торый обычно применяется для «тонкой» настройки приложения. Получить переменные среды, связанные с приложением, можно следующим образом:

```
LPSTR lpszVariable;  
LPVOID lpvEnv;  
lpvEnv = GetEnvironmentStrings();  
lpszVariable = (LPSTR)lpvEnv;  
while (lpszVariable[0] != '\0')  
{  
    SendMessage(hTextEnvaroment, LB_ADDSTRING, 0,  
(LPARAM) lpszVariable);  
    lpszVariable = lpszVariable + strlen(lpszVariable) + 1;  
}  
FreeEnvironmentStrings((LPTSTR)lpvEnv);
```

### ***Передача информации между процессами через переменные окружения***

Пользователь создает и инициализирует переменную среды, затем запускает приложение, и оно, анализируя блок, отыскивает одну или несколько «своих» переменных, а обнаружив, проверяет их значение и соответствующим образом настраивается. Инициализируем переменную среды до создания процесса:

```
_itoa((INT)hFileBin, envValue, 10);  
SetEnvironmentVariable("BinFileHeader", envValue);
```

Поле создания дочернего процесса отыскиваем в нем эту же переменную и считываем ее значение:

```
GetEnvironmentVariable("BinFileHeader", envVal, 16);  
envHandle = atoi(envVal);
```

### ***Диаграмма состояния процесса***

Каждый процесс в системе может находиться в различных состояниях (активен или неактивен), а также в процессе работы использует какие-либо ресурсы системы: время ЦП, оперативную или виртуальную память, создает или удаляет внутренние потоки и т. д. Система располагает средствами, позволяющими производить мониторинг таких действий со стороны процесса. Данные такого типа удобно отображать в виде временной диаграммы, что очень удобно для наблюдения за состоянием системных объектов.

## ***Список процессов***

Все современные ОС содержат средства (библиотеки, такие как Tlhelp32.dll) для наблюдения за состоянием системных объектов. В основном эти средства позволяют делать «снимок» их состояния на определенный момент времени.

```
hS = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
// делаем "снимок"
PROCESSENTRY32 P;
P.dwSize = sizeof(PROCESSENTRY32);
if (Process32First(hS, &P))
    do
        { // выводим элемент списка
            ...
        } while (Process32Next(hS, &P));
CloseHandle(hS); // удаляем снимок
```

## ***Идентификатор процесса***

Все процессы в системе имеют уникальные идентификаторы. Они позволяют системе различать все процессы. Пользователю такой идентификатор доступен только для чтения, он присваивается процессу на этапе его создания и не изменяется в течение всего времени работы процесса:

```
PROCESSENTRY32 P;
do
    { // выводим элемент списка
        _itoa(P.th32ProcessID, szStrVal, 10); // ID процесса
        _itoa(P.th32ParentProcessID, szStrVal, 10); // ID родителя
    } while (Process32Next(hS, &P));
```

## ***Кто родитель процесса***

Делая «снимок» состояния системных объектов (процессов), можно получить идентификатор родителя для процесса. Далее отыскав в этом же списке процессов процесс с таким же идентификатором, можно узнать его имя и другую информацию, касающуюся этого процесса (родителя):

```

PROCESSENTRY32 P;
do
{ // выводим элемент списка
_itoa(P.th32ParentProcessID, szStrVal, 10); // ID родителя
} while (Process32Next(hS, &P));

```

### ***Текущая рабочая директория процесса***

За каждым процессом закрепляется некая директория, являющаяся основной при открытии, сохранении и других операциях с файловой системой от имени этого процесса:

```

TCHAR szDir[255];
GetCurrentDirectory(255, szDir); // получаем
//текущую директорию
SetCurrentDirectory(szDir); // устанавливаем текущую
// директорию

```

### ***Время выполнения процесса***

На использование процессом ресурсов ОС затрачивается некоторое время, система же предоставляет некоторые функции для отображения этой информации (время создания, время ядра, время пользователя и др.).

```

FILETIME timeCreation, FILETIME timeExit;
FILETIME timeKernel, FILETIME timeUsed;
SYSTEMTIME timeSys;
GetProcessTimes(GetCurrentProcess(),
&timeCreation, &timeExit, &timeKernel, &timeUsed);
FileTimeToLocalFileTime(&timeCreation, &timeCreation);
FileTimeToSystemTime(&timeCreation, &timeSys);
SetWindowText(hProcTime, szTime);
FileTimeToSystemTime(&timeUsed, &timeSys); // Used Process
SetWindowText(hProcTimeUs, szTime);
FileTimeToSystemTime(&timeKernel, &timeSys);
// Kernel used Process
SetWindowText(hProcTimeKrn, szTime);

```

### ***Наследование дочерними процессами среды родительского процесса***

По умолчанию система наследует всем создаваемым процессом среду его родителя, но при его создании имеется возможность задать собственную среду для процесса, используя седьмой параметр в функции *CreateProcess(...)*, или указать NULL для наследования по умолчанию:

```
// Run process
    BOOL Res = CreateProcess (NULL, szRunAppl, NULL, NULL,
    TRUE, 0, pEnvaroment, NULL, &stInfo, &prTInfo);
```

## Литература

1. Рихтер, Дж. Windows для профессионалов: создание эффективных Win32 приложений с учетом специфики 64-разрядной версии Windows / Дж. Рихтер; пер. с англ. – 4-е изд. – СПб.: Питер; М.: Издательско-торговый дом «Русская редакция», 2001. – 752 с.
2. Шилдт, Г. Полный справочник по C++ / Г. Шилдт. – 4-е изд. – М.: Вильямс, 2006. – 796 с.
3. Петзолд, Ч. Программирование для Windows 95: в 2 т. / Ч. Петзолд; пер. с англ. – СПб.: BHV – Санкт-Петербург, 1997. – Т. 2. – 368 с.
4. Гордеев А.В. Системное программное обеспечение / А.В. Гордеев, А.Ю. Молчанов. – СПб.: Питер, 2003. – 736 с.
5. Румянцев, П.В. Азбука программирования в WIN32 API / П.В. Румянцев. – СПб.: Питер, 2004. – 310 с.
6. Разработка приложений на Microsoft Visual C++ 6.0. Учебный курс: официальное пособие Microsoft для самостоятельной подготовки / пер. с англ. – М.: Издательско-торговый дом «Русская редакция», 2000. – 576 с.
7. Круглински, Д. Программирование на Microsoft Visual C++ 6.0 для профессионалов / Д. Круглинский, С. Уингоу, Дж. Шеферд; пер. с англ. – СПб.: Питер; М.: Издательско-торговый дом «Русская редакция», 2004. – 861 с.
8. Фролов, А. Операционная система MS-DOS: в 2 т. / А. Фролов, Г. Фролов. – М.: Диалог-МИФИ, 1992. – Т. 1, кн. 3. – 222 с.
9. Джонсон, М. Харт. Системное программирование в среде Windows / М. Харт Джонсон. – М.: Издательский дом «Вильямс», 2001. – 464 с.
10. Рихтер, Дж. Программирование серверных приложений для Microsoft Windows 2000. Мастер-класс / Дж. Рихтер, Дж. Д. Кларк; пер. с англ. – СПб.: Питер; М.: Издательско-торговый дом «Русская редакция», 2001. – 592 с.

Учебное издание

## ОПЕРАЦИОННЫЕ СИСТЕМЫ И СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Методические указания к лабораторным работам  
для студентов специальностей 1-40 01 01  
«Программное обеспечение информационных технологий»  
и 1-40 01 02 «Информационные системы и технологии»

Составитель  
РАЗОРЁНОВ Николай Александрович

Редактор Т.Н. Микулик  
Компьютерная верстка А.Г. Занкевич

---

Подписано в печать 12.09.2011.

Формат 60×84<sup>1</sup>/<sub>16</sub>. Бумага офсетная.

Отпечатано на ризографе. Гарнитура Таймс.

Усл. печ. л. 5,46. Уч.-изд. л. 4,27. Тираж 100. Заказ 1195.

---

Издатель и полиграфическое исполнение:  
Белорусский национальный технический университет.

ЛИ № 02330/0494349 от 16.03.2009.

Проспект Независимости, 65. 220013, Минск.