**Основы программирования для Win32 API**

[*(с) 2004, Алексей Соловьев*](http://lab127.karelia.ru/email.php?eml_ref=avs)

*Материал рассматривается на примере пакета Borland C++ 5.5 command line tools*

1. **Стартовая функция WinMain**

Программа на Си для Windows, как и для любой другой платформы, должна обязательно содержать некоторую "стартовую" функцию, которой передается управление при запуске программы. Вообще говоря, имя такой "стартовой" функции может различаться в различных компиляторах, но исторически сложилось так (а, кроме того, имеются еще и стандарты ANSI и ISO, к которым, правда, производители коммерческих компиляторов типа Microsoft и Borland/Inprise относятся без особого трепета), что такой функцией является:

int **main**()

У этой функции может быть до трех параметров:

int **main**(int argc, char \*argv[], char \*env[])

* + argc - количество параметров в командной строке (включая имя программы),
  + argv - массив строк-параметров (argv[0] - имя программы),
  + env - массив строк-переменных окружения.

Многие компиляторы для Windows "понимают" такую стартовую функцию. Однако при этом они создают хотя и 32-битное, но консольное приложение. Пример 1 (example1.cpp):

#include <stdio.h>

int main() {

printf("Hello, world!");

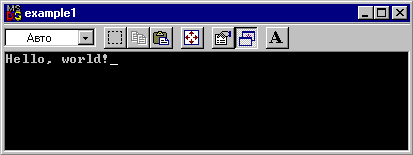
getc(stdin);

return 0;

}

Компилируем:

bcc32 example1.cpp

Запускаем:  


При использовании стандартных библиотек (stdio, stdlib и т. п.) вам не потребуется никаких лишних телодвижений по сравнению с обычными методами написания программ на Си. Если же ваша цель - 32-битное приложение с графическим интерфейсом, то черное консольное окошко будет вас раздражать. Пример (example2.cpp):

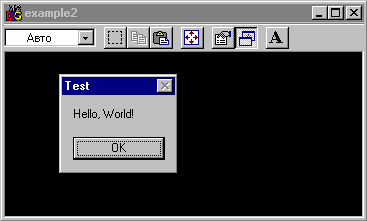
#include <windows.h>

int main() {

MessageBox(NULL,"Hello, World!","Test",MB\_OK);

return 0;

}



В общем, чтобы получить нормальное приложение без каких-либо "довесков" типа консольного окошка, используется другая стартовая функция:

int WINAPI WinMain (HINSTANCE hInst, HINSTANCE hpi, LPSTR cmdline, int ss)

* + hInst - дескриптор для данного экземпляра программы,
  + hpi - в Win32 не используется (всегда NULL),
  + cmdline - командная строка,
  + ss - код состояния главного окна.

Пример (example3.cpp):

#include <windows.h>

int WINAPI WinMain(HINSTANCE,HINSTANCE,LPSTR,int) {

MessageBox(NULL,"Hello, World!","Test",MB\_OK);

return 0;

}

Кроме того, компилятору и компоновщику нужно сообщить о том, что вы делаете графическое приложение. Для bcc32 это опция *-tW*:

bcc32 -tW example3.cpp



Если же вы соскучитесь по черному консольному окошку, его можно в любой момент создать при помощи вызова Win32 API

BOOL **AllocConsole**(void)

1. **О типах, о функциях**

Как известно, в Си есть лишь три базовых типа (char, int, float/double) и еще несколько их вариаций с модификаторами signed/unsigned, short/long. Однако фирме Microsoft зачем-то понадобилось описывать функции Win32 API с помощью переопределенных типов:

typedef unsigned char **BYTE**;

typedef unsigned short **WORD**;

typedef unsigned int **UINT**;

typedef int **INT**;

typedef long **BOOL**;

#define **FALSE** 0

#define **TRUE** 1

typedef long **LONG**;

typedef unsigned long **DWORD**;

typedef void \***LPVOID**;

typedef char **CHAR**;

typedef CHAR \***LPSTR**;

typedef const CHAR \***LPCSTR**;

Кроме перечисленных простых типов, практически ни один вызов Win32 API не обходится без "штучек" с "ручками" - переменных типа *handle* ("ручка"), которые идентифицируют некоторый объект ("штучку"). Такие "ручки" принято называть дескрипторами. Реально такая переменная представляет собой всего лишь указатель на некоторую системную структуру или индекс в некоторой системной таблице.

typedef void \***HANDLE**; /\* абстрактный дескриптор (например, файла) \*/

typedef void \***HMODULE**; /\* дескриптор модуля \*/

typedef void \***HINSTANCE**; /\* дескриптор экземпляра программы \*/

typedef void \***HKEY**; /\* дескриптор ключа в реестре \*/

typedef void \***HGDIOBJ**; /\* дескриптор графического примитива (перо, шрифт, кисть, палитра,...) \*/

typedef void \***HWND**; /\* дескриптор окна \*/

typedef void \***HMENU**; /\* дескриптор меню \*/

typedef void \***HICON**; /\* дескриптор иконки \*/

typedef void \***HBITMAP**; /\* дескриптор картинки \*/

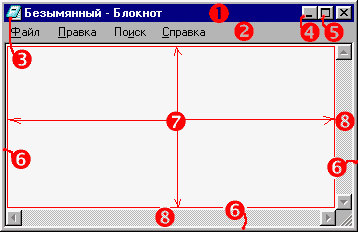
typedef void \***HFONT**; /\* дескриптор шрифта \*/

При заполнении различных структур часто требуется указать такую "ручку" от какой-нибудь "штучки". Очень часто вместо конкретного дескриптора допустимо передавать значение NULL, означающее, что вы еще не обзавелись такой "штучкой" или собираетесь использовать "штучку" по умолчанию.

В стандартных версиях Си для функций используются два варианта соглашения о передаче параметров: соглашение языка Си (параметры функции помещаются в стек в порядке обратном их описанию, очистку стека производит вызывающая процедура) и соглашение языка Паскаль (параметры функции помещаются в стек в (прямом) порядке их описания, очистку стека производит вызываемая процедура). Для этих соглашений использовались, соответственно, модификаторы cdecl и pascal. При описании функций Win32 API используется модификатор WINAPI, а для описания пользовательских функций обратного вызова - модификатор CALLBACK. Оба этих модификатора являются переопределением специального модификатора \_stdcall, соответствующего соглашению о передаче параметров, использующегося исключительно в Win32 API, - *Standard Calling Convention* (параметры функции помещаются в стек в порядке обратном их описанию, очистку стека производит вызываемая процедура).

1. **Окна**

Окно - это прямоугольная область экрана, в котором приложение отображает информацию и получает реакцию от пользователя. Одновременно на экране может отображаться несколько окон, в том числе, окон других приложений, однако лишь одно из них может получать реакцию от пользователя - *активное окно*. Пользователь использует клавиатуру, мышь и прочие устройства ввода для взаимодействия с приложением, которому принадлежит активное окно. Каждое 32-битное приложение создает, по крайней мере, одно окно, называемое *главным окном*, которое обеспечивает пользователя основным интерфейсом взаимодействия с приложением.

Окно приложения может содержать строку заголовка *title bar* (1), строку меню *menu bar* (2), системное меню *system menu* (3), кнопку сворачивания окна *minimize box* (4), кнопку разворачивания окна *maximize box* (5), рамку изменения размеров *sizing border* (6), клиентскую область *client area* (7), горизонтальную и вертикальную полосы прокрутки *scroll bars* (8):  


Меню, строка заголовка с системными кнопками, системное меню, рамка изменения размеров и полосы прокрутки относятся к области окна, называемой *неклиентской областью* (*non-client area*). С неклиентской областью Windows справляется сама, а вот за содержимое и обслуживание клиентской области отвечает приложение.

Кроме главного окна, приложение может использовать еще и другие типы окон: управляющие элементы (*controls*), диалоговые окна (*dialog boxes*), окна-сообщения (*message boxes*). *Управляющий элемент* - окно, непосредственно обеспечивающее тот или иной способ ввода информации пользователем. К управляющим элементам относятся: кнопки, поля ввода, списки, полосы прокрутки и т.п. Управляющие элементы обычно не болтаются сами по себе, а проживают в каком-либо диалоговом окне. *Диалоговое окно* - это временное окно, напичканное управляющими элементами, обычно использующееся для получения дополнительной информации от пользователя. Диалоговые окна бывают модальные (*modal*) и немодальные (*modeless*). Модальное диалоговое окно требует, чтобы пользователь обязательно ввел обозначенную в окне информацию и закрыл окно прежде, чем приложение продолжит работу. Немодальное диалоговое окно позволяет пользователю, не закрывая диалогового окна, переключаться на другие окна этого приложения. *Окно-сообщение* - это диалоговое окно предопределенного системой формата, предназначенное для вывода небольшого текстового сообщения с одной или несколькими кнопками. Пример такого окна показан в [Примере 3](http://dims.karelia.ru/win32/#ex3).

В отличие от традиционного программирования на основе линейных алгоритмов, программы для Windows строятся по принципам событийно-управляемого программирования (*event-driven programming*) - стиля программирования, при котором поведение компонента системы определяется набором возможных внешних событий и ответных реакций компонента на них. Такими компонентами в Windows являются окна. С каждым окном в Windows связана определенная функция обработки событий. События для окон называются *сообщениями*. Сообщение относится к тому или иному типу, идентифицируемому определенным кодом (32-битным целым числом), и сопровождается парой 32-битных параметров (WPARAM и LPARAM), интерпретация которых зависит от типа сообщения. В заголовочном файле windows.h для кодов сообщений определены константы с интуитивно понятными именами:

#define WM\_CREATE 0x0001 /\* сообщение о создании окна \*/

#define WM\_DESTROY 0x0002 /\* сообщение об уничтожении окна \*/

#define WM\_SIZE 0x0005 /\* сообщение об изменении размеров окна \*/

#define WM\_COMMAND 0x0111 /\* сообщение от команды меню или управляющего элемента \*/

Таким образом, задача любого приложения - создать главное окно и сообщить Windows функцию обработки событий для этого окна. Все самое интересное для приложения будет происходить именно в функции обработки событий главного окна.

Для стандартных управляющих элементов (библиотека Common Controls Library - COMCTL32.DLL) в Windows имеются предопределенные обработчики событий, которые при наступлении интересных событий сообщают всяческую полезную информацию окну, содержащему этот управляющий элемент. Стандартная библиотека Common Dialog Box Library (COMDLG32.DLL) содержит несколько готовых весьма полезных диалоговых окон с обработчиками: диалоги выбора файла, настроек печати, выбора шрифта, выбора цвета и др. Кроме того, любая среда разработки (VisualBasic, Delphi, VisualC++ и т.п.) навязывает разработчику дополнительный набор готовых управляющих элементов и диалогов - иногда достаточно удобных, иногда не очень.

1. **Структура программы**

Программа для Win32 обычно состоит из следующих блоков:

|  |
| --- |
| #include <windows.h>  int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR cmdline,int ss) {  /\* ***Блок инициализации:***  *создание класса главного окна,*  *создание главного окна,*  *загрузка ресурсов и т.п.* \*/  /\* ***Цикл обработки событий:*** \*/  MSG msg;  while (GetMessage(&msg,(HWND)NULL,0,0)) {  TranslateMessage(&msg);  DispatchMessage(&msg);  }  return msg.wParam;  }  LRESULT CALLBACK MainWinProc(HWND hw,UINT msg,WPARAM wp,LPARAM lp) {  /\* ***Обработка сообщений главного окна*** \*/  switch (msg) {  case WM\_CREATE:  /\* ... \*/  return 0;  case WM\_COMMAND:  /\* ... \*/  return 0;  case WM\_DESTROY:  /\* ... \*/  PostQuitMessage(0);  return 0;  /\* ... \*/  }  return DefWindowProc(hw,msg,wp,lp);  } |

Каждое окно принадлежит определенному классу окон. Окна одного класса имеют схожий вид, обслуживаются общей процедурой обработки событий, имеют одинаковые иконки и меню. Обычно каждое приложение создает для главного окна программы свой класс. Если приложению требуются дополнительные нестандартные окна, оно регистрирует другие классы. Стандартные диалоги и управляющие элементы принадлежат к предопределенным классам окон, для них не надо регистрировать новые классы. Чтобы определить новый класс окон, надо заполнить структуру WNDCLASS, содержащую следующие поля:

* + UINT style - стиль (поведение) класса окон,
  + WNDPROC lpfnWndProc - процедура обработки событий окна,
  + int cbClsExtra - размер дополнительной памяти в системной структуре класса для данных пользователя,
  + int cbWndExtra - размер дополнительной памяти в системной структуре окна для данных пользователя,
  + HINSTANCE hInstance - дескриптор модуля (экземпляра программы), в котором реализована процедура обработки,
  + HICON hIcon - дескриптор иконки окна,
  + HCURSOR hCursor - дескриптор курсора мыши для окна,
  + HBRUSH hbrBackground - дескриптор "кисточки" для закрашивания фона окна,
  + LPCSTR lpszMenuName - имя ресурса, содержащего меню окна,
  + LPCSTR lpszClassName - имя класса.

Класс регистрируется при помощи функции:

WORD WINAPI **RegisterClass**(const WNDCLASS \*lpwc)

При успешном завершении функция возвращает целочисленный код, соответствующий строке-имени класса в общесистемной таблице строк (такой код называется *атомом*). При ошибке возвращается 0.

Для создания окна вызывается функция:

HWND WINAPI **CreateWindow**(

LPCSTR lpClassName, /\* имя класса \*/

LPCSTR lpWindowName, /\* имя окна (заголовок) \*/

DWORD dwStyle, /\* стиль (поведение) окна \*/

int x, /\* горизонтальная позиция окна на экране \*/

int y, /\* вертикальная позиция окна на экране \*/

int nWidth, /\* ширина окна \*/

int nHeight, /\* высота окна \*/

HWND hWndParent, /\* дескриптор родительского окна \*/

HMENU hMenu, /\* дескриптор меню \*/

HANDLE hInstance, /\* дескриптор экземпляра программы \*/

LPVOID lpParam /\* указатель на какую-нибудь ерунду \*/

)

Вместо параметров x, y, nWindth, nHeight допустимо передавать константу CW\_USEDEFAULT, позволяющую операционной системе задать эти числа по ее усмотрению.

Интерпретация кода стиля определяется классом окна. Стиль определяет не только оформление окна, но и его поведение. Общие для всех классов константы стилей (при необходимости объединяются операцией побитовое ИЛИ):

* + WS\_DISABLED - при создании окно заблокировано (не может получать реакцию от пользователя);
  + WS\_VISIBLE - при создании окно сразу же отображается (не надо вызывать **ShowWindow**);
  + WS\_CAPTION - у окна есть строка заголовка;
  + WS\_SYSMENU - у окна есть системное меню;
  + WS\_MAXIMIZEBOX - у окна есть кнопка разворачивания;
  + WS\_MINIMIZEBOX - у окна есть кнопка сворачивания;
  + WS\_SIZEBOX или WS\_THICKFRAME - у окна есть рамка изменения размеров;
  + WS\_BORDER - у окна есть рамка (не подразумевает изменение размеров);
  + WS\_HSCROLL или WS\_VSCROLL - у окна есть горизонтальная или вертикальная прокрутка;
  + WS\_OVERLAPPED или WS\_TILED - "перекрываемое" окно - обычное окно с рамкой и строкой заголовка;
  + WS\_POPUP - "всплывающее" окно;
  + WS\_OVERLAPPEDWINDOW - "перекрываемое" окно с системным меню, кнопками сворачивания/разворачивания, рамкой изменения размеров, короче, типичный стиль для главного окна приложения.

Во время выполнения функции **CreateWindow** процедуре обработки событий окна посылается сообщение WM\_CREATE. При успешном выполнении функции возвращается дескриптор созданного окна, при неудаче - NULL.

После создания окна неплохо бы сделать его видимым (отобразить), если только оно не создано со стилем WS\_VISIBLE:

BOOL WINAPI **ShowWindow**(HWND hw, int ss)

Второй параметр этой функции - код состояния отображения окна. В качестве этого кода можно взять значение четвертого параметра, с которым была запущена функция **WinMain**. Другие возможные значения этого параметра:

* + SW\_SHOW - отобразить и активировать окно;
  + SW\_HIDE - скрыть окно;
  + SW\_MAXIMIZE - развернуть окно на весь экран;
  + SW\_RESTORE - активировать окно и отобразить его в размерах по умолчанию;
  + SW\_MINIMIZE - свернуть окно.

Если перед вызовом этой функции окно было видимым, функция возвращает TRUE, если же окно было скрыто - FALSE.

Если клиентская область главного окна приложения содержит объекты, прорисовываемые по сообщению WM\_PAINT, имеет смысл прорисовать эти объекты сразу после отображения главного окна на экране. Функция **UpdateWindow** непосредственно вызывает процедуру обработки событий указанного окна с сообщением WM\_PAINT (минуя очередь сообщений приложения):

BOOL WINAPI **UpdateWindow**(HWND hw)

Windows использует два способа доставки сообщений процедуре обработки событий окна:

* + непосредственный вызов процедуры обработки событий (*внеочередные* или *неоткладываемые сообщения* - *nonqueued messages*);
  + помещение сообщения в связанный с данным приложением буфер типа FIFO, называемый *очередью сообщений* - *message queue* (*откладываемые сообщения* - *queued messages*).

К внеочередным сообщениям относятся те сообщения, которые непосредственно влияют на окно, например, сообщение активации окна WM\_ACTIVATE и т.п. Кроме того, вне очереди сообщений обрабатываются сообщения, сгенерированные различными вызовами Win32 API, такими как **SetWindowPos**, **UpdateWindow**, **SendMessage**, **SendDlgItemMessage**...

К откладываемым сообщениям относятся сообщения, связанные с реакцией пользователя: нажатие клавиш на клавиатуре, движение мышки и клики. Чтобы извлечь сообщение из очереди, программа вызывает функцию

BOOL WINAPI **GetMessage**(

MSG \*lpmsg, /\* сюда попадает сообщение со всякими параметрами \*/

HWND hw, /\* извлекать только сообщения для указанного окна (NULL - все) \*/

UINT wMsgFilterMin, /\* фильтр сообщений (нам не надо - ставим 0) \*/

UINT wMsgFilterMax /\* фильтр сообщений (нам не надо - ставим 0) \*/

)

Эта функция возвращает FALSE, если получено сообщение WM\_QUIT, и TRUE в противном случае. Очевидно, что условием продолжения цикла обработки событий является результат этой функции. Если приложение хочет завершить свою работу, оно посылает само себе сообщение WM\_QUIT при помощи функции

void WINAPI **PostQuitMessage**(int nExitCode)

Ее параметр - статус выхода приложения. Обычно эта функция вызывается в ответ на сообщение об уничтожении окна WM\_DESTROY.

После извлечения сообщения из очереди следует вызвать функцию **TranslateMessage**, переводящую сообщения от нажатых клавиш в удобоваримый вид, а затем **DispatchMessage**, которая определяет предназначенное этому сообщению окно и вызывает соответствующую процедуру обработки событий.

BOOL WINAPI **TranslateMessage**(const MSG \*lpmsg)

LONG WINAPI **DispatchMessage**(const MSG \*lpmsg)

Результат возврата соответствует значению, которое вернула процедура обработки событий (обычно никому не нужен).

Процедура обработки сообщений окна должна быть объявлена по следующему прототипу:

LRESULT CALLBACK *WindowProc*(HWND hw,UINT msg,WPARAM wp,LPARAM lp)

Значения параметров: hw - дескриптор окна, которому предназначено сообщение, msg - код сообщения, wp и lp - 32-битные параметры сообщения, интерпретация которых зависит от кода сообщения. Зачастую старший/младший байт или старшее/младшее слово параметров сообщения несут независимый смысл, тогда удобно использовать определенные в windows.h макросы:

#define **LOBYTE**(w) ((BYTE) (w))

#define **HIBYTE**(w) ((BYTE) (((WORD) (w) >> 8) & 0xFF))

#define **LOWORD**(l) ((WORD) (l))

#define **HIWORD**(l) ((WORD) (((DWORD) (l) >> 16) & 0xFFFF))

Например, сообщение WM\_COMMAND посылается окну в трех случаях:

* + пользователь выбрал какую-либо команду меню;
  + пользователь нажал "горячую" клавишу (*accelerator*);
  + в дочернем окне произошло определенное событие.

При этом параметры сообщения интерпретируются следующим образом. Старшее слово параметра WPARAM содержит: 0 в первом случае, 1 во втором случае и код события в третьем случае. Младшее слово WPARAM содержит целочисленный идентификатор пункта меню, "горячей" клавиши или дочернего управляющего элемента. Параметр LPARAM в первых двух случаях содержит NULL, а в третьем случае - дескриптор окна управляющего элемента.

Процедура обработки событий должна вернуть определенное 32-битное значение, интерпретация которого также зависит от типа сообщения. В большинстве случаев, если сообщение успешно обработано, процедура возвращает значение 0.

Процедура обработки событий не должна игнорировать сообщения. Если процедура не обрабатывает какое-то сообщение, она должна вернуть его системе для обработки по умолчанию. Для этого вызывается функция:

LRESULT WINAPI **DefWindowProc**(HWND hw, UINT msg, WPARAM wp, LPARAM lp)

Все описанное в данном параграфе суммируется в примере 4 (example4.cpp):

#include <windows.h>

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

#define ID\_MYBUTTON 1 /\* *идентификатор для кнопочки внутри главного окна* \*/

int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int ss) {

/\* *создаем и регистрируем класс главного окна* \*/

WNDCLASS wc;

wc.style=0;

wc.lpfnWndProc=MainWinProc;

wc.cbClsExtra=wc.cbWndExtra=0;

wc.hInstance=hInst;

wc.hIcon=NULL;

wc.hCursor=NULL;

wc.hbrBackground=(HBRUSH)(COLOR\_WINDOW+1);

wc.lpszMenuName=NULL;

wc.lpszClassName="Example 4 MainWnd Class";

if (!RegisterClass(&wc)) return FALSE;

/\* *создаем главное окно и отображаем его* \*/

HWND hMainWnd=CreateWindow("Example 4 MainWnd Class","EXAMPLE 4",WS\_OVERLAPPEDWINDOW,

CW\_USEDEFAULT,CW\_USEDEFAULT,CW\_USEDEFAULT,CW\_USEDEFAULT,NULL,NULL,hInst,NULL);

if (!hMainWnd) return FALSE;

ShowWindow(hMainWnd,ss);

UpdateWindow(hMainWnd);

MSG msg; /\* *цикл обработки событий* \*/

while (GetMessage(&msg,NULL,0,0)) {

TranslateMessage(&msg);

DispatchMessage(&msg);

}

return msg.wParam;

}

/\* *процедура обработки сообщений для главного окна* \*/

LRESULT CALLBACK MainWinProc(HWND hw,UINT msg,WPARAM wp,LPARAM lp) {

switch (msg) {

case WM\_CREATE:

/\* *при создании окна внедряем в него кнопочку* \*/

CreateWindow("button","My button",WS\_CHILD|BS\_PUSHBUTTON|WS\_VISIBLE,

5,5,100,20,hw,(HMENU)ID\_MYBUTTON,NULL,NULL);

/\* *стиль WS\_CHILD означает, что это дочернее окно и для него*

*вместо дескриптора меню будет передан целочисленный идентификатор,*

*который будет использоваться дочерним окном для оповещения*

*родительского окна через WM\_COMMAND* \*/

return 0;

case WM\_COMMAND:

/\* *нажата наша кнопочка?* \*/

if ((HIWORD(wp)==0) && (LOWORD(wp)==ID\_MYBUTTON))

MessageBox(hw,"You pressed my button","MessageBox",MB\_OK|MB\_ICONWARNING);

return 0;

case WM\_DESTROY:

/\* *пользователь закрыл окно, программа может завершаться* \*/

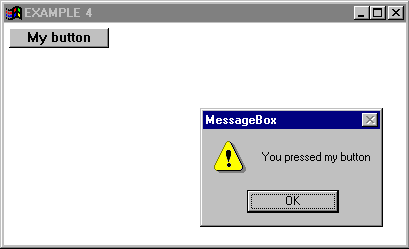
PostQuitMessage(0);

return 0;

}

return DefWindowProc(hw,msg,wp,lp);

}

Приведенный пример создает окно с кнопкой "My button", при нажатии на которую вылезает окно-сообщение:  


1. **Ресурсы**

*Ресурсы* - это бинарные данные, добавляемые в исполняемый файл при компоновке программы. К стандартным ресурсам относятся: иконки, курсоры, меню, диалоги, растровые изображения (BMP), векторные изображения (EMF), шрифты, таблицы горячих клавиш, таблицы строк, информация о версии программы или модуля. В процессе разработки программы ресурсы описывают в отдельном текстовом файле - *файле описания ресурсов* (расширение .rc), - а затем при помощи компилятора ресурсов переводят в бинарный вид и добавляют в исполняемый файл на этапе компоновки исполняемого файла. Использование ресурсов значительно облегчает работу программиста по визуализации графических примитивов интерфейса программы.

Файл описания ресурсов состоит из операторов, объединяемых в блоки. Один оператор занимает одну строку файла. Допускается использовать комментарии, определяемые так же, как в программе на языке Си. Файл описания ресурсов перед компиляцией так же обрабатывается препроцессором, поэтому в нем можно использовать директивы препроцессора (#include, #define, ...) и макроопределения. В сложных "блочных" описаниях ресурсов вместо ключевых слов BEGIN и END можно использовать { и }, соответственно.

Иконки, картинки и курсоры мыши можно описать двумя способами (квадратные скобки не являются частью синтаксиса оператора и означают необязательный элемент):

nameID *RESOURCETYPE* [load-option] [mem-option] filename

Здесь nameID - численный или строковой идентификатор; *RESOURCETYPE* - ключевое слово, обозначающее тип ресурса: **ICON**, **BITMAP** или **CURSOR**; load-option и mem-option - всякие неинтересные в данный момент опции, которые можно спокойно пропустить; filename - имя файла, содержащее соответствующий ресурс.

Примеры:

disk1 **BITMAP** "disk.bmp"

12 **ICON** "myicon.ico"

Эти ресурсы можно внедрить в виде шестнадцатеричных кодов непосредственно в файл ресурсов:

nameID *RESOURCETYPE*

BEGIN

*hex data*

END

Пример:

FltBmp **BITMAP**

{

'42 4D A2 00 00 00 00 00 00 00 3E 00 00 00 28 00'

'00 00 19 00 00 00 19 00 00 00 01 00 01 00 00 00'

'00 00 64 00 00 00 00 00 00 00 00 00 00 00 00 00'

'00 00 00 00 00 00 00 00 00 00 FF FF FF 00 FF FF'

'FF 80 FF FF FF 80 FF FF FF 80 FF FF FF 80 FF FF'

'FF 80 FF FF FF 80 FF FF FF 80 C0 FF 81 80 FE FF'

'BF 80 FE FF BF 80 FE FF BF 80 FE FF BF 80 FE FF'

'BF 80 FE FF BF 80 FE FF BF 80 FE FF BF 80 FE FF'

'BF 80 FE FF BF 80 FE 00 3F 80 FF FF FF 80 FF FF'

'FF 80 FF FF FF 80 FF FF FF 80 FF FF FF 80 FF FF'

'FF 80'

}

Следует отметить, что первая иконка в ресурсах будет использоваться "Проводником" как иконка исполняемого файла.

Меню описывается следующим образом:

nameID **MENU** [load-option] [mem-option]

BEGIN

*item-definitions*

...

END

Здесь *item-definitions* - один из трех операторов:

**MENUITEM** text, result [, optionlist]

/\* обычный пункт меню \*/

**MENUITEM SEPARATOR**

/\* строка-сепаратор \*/

**POPUP** text [, optionlist]

BEGIN /\* подменю \*/

*item-definitions*

...

END

Параметры операторов имеют следующий смысл: text - текст пункта меню или подменю (может содержать комбинации **\t** - табуляция, **\a** - выравнивание по правому краю, **&** - следующий символ подчеркивается, обозначает "горячую" клавишу для указанного пункта меню); result - целочисленный идентификатор пункта меню, посылаемый окну-владельцу через сообщение WM\_COMMAND при выборе этого пункта меню; optionlist - необязательный список опций, разделенных запятой или пробелом:

* + CHECKED - рядом с пунктом меню отображается галочка,
  + GRAYED - пункт меню неактивен (не может быть выбран) и отображается серым цветом и др.

Доступ к ресурсам, скомпонованным с исполняемым файлом, можно получить при помощи следующих функций:

HICON WINAPI **LoadIcon**(HINSTANCE hInst, LPCSTR lpIconName)

HBITMAP WINAPI **LoadBitmap**(HINSTANCE hInst, LPCSTR lpBitmapName)

HCURSOR WINAPI **LoadCursor**(HINSTANCE hInst, LPCSTR lpCursorName)

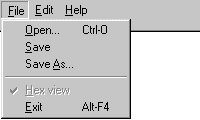
HMENU WINAPI **LoadMenu**(HINSTANCE hInst, LPCSTR lpMenuName)

Первый параметр этих функций - дескриптор экземпляра программы, второй - идентификатор соответствующего ресурса. Если ресурс идентифицируется не именем, а числом, то следует использовать макрос, объявленный в windows.h:

#define **MAKEINTRESOURCE**(i) (LPSTR) ((DWORD) ((WORD) (i)))

Например:

HMENU hMainMenu=LoadMenu(hInst,MAKEINTRESOURCE(10));

Для закрепления полученных сведений, давайте добавим к [примеру 4](http://dims.karelia.ru/win32/#ex4) какую-нибудь иконку и такое меню:  


Для этого создаем файл ресурсов (example4a.rc):

Ex4\_Icon **ICON** "[myicon.ico](http://dims.karelia.ru/win32/myicon.ico)"

Ex4\_Menu **MENU**

{

**POPUP** "&File"

{

**MENUITEM** "&Open...\tCtrl-O", 2

**MENUITEM** "&Save", 3

**MENUITEM** "Save &As...", 4

**MENUITEM** SEPARATOR

**MENUITEM** "&Hex view", 5, CHECKED GRAYED

**MENUITEM** "&Exit\tAlt-F4", 6

}

**POPUP** "&Edit"

{

**MENUITEM** "&Copy", 7

**MENUITEM** "&Paste", 8

**POPUP** "Popup"

{

**MENUITEM** "1", 9

**MENUITEM** "2", 10

**MENUITEM** "3", 11

}

**MENUITEM SEPARATOR**

**MENUITEM** "Search", 12

}

**POPUP** "&Help"

{

**MENUITEM** "&About...\tF1", 13

}

}

Для перевода файла описания ресурсов в бинарный вид используется компилятор ресурсов *Borland Resource Compiler*:

brcc32 example4a.rc

В результате получается файл example4a.res, который потребуется в процессе компоновки.

В [примере 4](http://dims.karelia.ru/win32/#ex4) надо изменить строки

wc.hIcon=NULL;

wc.lpszMenuName=NULL;

на

wc.hIcon=LoadIcon(hInst,"Ex4\_Icon");

wc.lpszMenuName="Ex4\_Menu";

Чтобы программа не была такой скучной, изменим обработчик сообщения WM\_COMMAND:

case WM\_COMMAND:

if (HIWORD(wp)==0) {

char buf[256];

switch (LOWORD(wp)) {

case 6: /\* *команда меню Exit* \*/

PostQuitMessage(0);

default: /\* *все остальные команды* \*/

wsprintf(buf,"Command code: %d",LOWORD(wp));

MessageBox(hw,buf,"MessageBox",MB\_OK|MB\_ICONINFORMATION);

}

}

return 0;

В результате при выборе того или иного пункта меню выводится окно-сообщение с кодом команды.

Обратите внимание: среди команд меню не используется код 1, который отведен кнопке "My button". Это типичная практика назначать всем дочерним элементам окна и командам меню разные численные идентификаторы, что облегчает обработку сообщения WM\_COMMAND.

1. **Компиляция и компоновка сложных проектов**

Теперь компоновка программы будет более сложной, поэтому bcc32 с этой задачей не справится. В этом примере компилятор будет использоваться только для компилирования (запускаем с ключом *-с*):

bcc32 -c -tW example4a.cpp

В результате получаем объектный файл example4a.obj.

Чтобы собрать все части программы вместе, придется запускать компоновщик вручную (ilink32 или tlink32). В командной строке компоновщика указываются следующие параметры:

**ilink32** [options] objfiles,exefile,mapfile,libfiles,deffile,resfiles

* + options - о допустимых опциях можно узнать, запустив компоновщик без параметров. Нам потребуются:
    - *-aa* - тип приложения "графическое для Win32" (другие варианты: *-ap* - консольное приложение, *-ad* - драйвер);
    - *-Tpe* - формат выходного файла ".EXE" (другой вариант: *-Tpd* - ".DLL");
    - *-L путь* - путь для поиска библиотек и объектных файлов (обычно: *-Lc:\bcc55\lib*).
  + objfiles - список объектных файлов, из которых составляется программа, разделенных пробелом или знаком "+". Этот список должен начинаться с борландовского инициализационного объектного файла: c0w32.obj - для графического приложения под Win32 или c0x32.obj - для консольного приложения.
  + exefile - имя исполняемого файла, который получится в результате компоновки.
  + mapfile - имя файла, который после компиляции будет содержать карту сегментов вашей программы (оно вам надо? если нет, здесь делаем "пусто", а в опциях указываем *-x*, чтобы компоновщик не замусоривал рабочий каталог этой фигней).
  + libfiles - список библиотек, в которых надо искать не определенные в программе функции, разделенных пробелом или знаком "+". Как минимум, надо указать import32.lib, которая содержит код подключения к стандартным библиотекам Win32 API: kernel32.dll, user32.dll, gdi32.dll, advapi32.dll и др. Если вы используете какие-либо функции стандартных библиотек языка Си (stdlib, stdio, ...), надо указать еще cw32.lib.
  + deffile - файл параметров модуля (*module definition file*). Это текстовый файл, в котором определяются различные настройки компилируемой программы типа: размеры сегментов стека, кода, данных, "заглушка" (что будет происходить при попытке запуска программы в DOS) и проч. Если не указывать этот файл, компоновщик выберет вполне приличные для большинства случаев параметры по умолчанию.
  + resfiles - файлы ресурсов (разделяются пробелом или знаком "+").

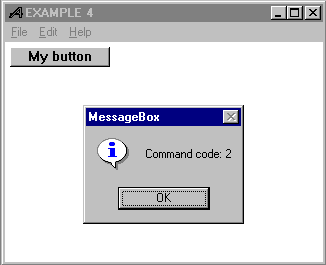
Компоновщик ilink32 умеет работать в пошаговом (инкрементирующем) режиме *incremental linking*, при этом он создает несколько файлов состояний (\*.IL?). При последующих попытках компиляции он использует их, так что процесс компиляции занимает меньше времени. Чтобы отключить пошаговую компиляцию и не замусоривать рабочий каталог этими файлами, следует указать опцию *-Gn*. Например, если при отключенной пошаговой компиляции программа компилируется 8 секунд, то первая компиляция в пошаговом режиме займет 25 секунд, а все последующие - не более 2 секунд.

Итак, компонуем модифицированный пример 4:

ilink32 -aa -Tpe -Lc:\bcc55\lib c0w32.obj+example4a.obj,example4a.exe,,import32.lib+cw32.lib,,example4a.res

Если все сделано правильно, первое, что становится сразу заметным - у исполняемого файла появилась иконка:

|  |  |
| --- | --- |
| http://dims.karelia.ru/win32/prop4.gif | http://dims.karelia.ru/win32/prop4a.gif |

Эта же иконка отображается в строке заголовка главного окна программы. Под строкой заголовка отображается созданное нами меню. При выборе любой команды меню появляется окно-сообщение с кодом команды. При выборе команды "Exit" программа завершается.  


Если ваш проект состоит из множества файлов, то компилировать и компоновать их вручную становится затруднительно. В таких случаях используются сценарии компиляции, которые обрабатываются программой make. Сценарий компиляции - текстовый файл с именем Makefile, описывающий зависимости между различными файлами проекта, следующего формата:

правило: зависимости

команды для выполнения

Сценарий компиляции должен содержать как минимум одно правило. Строка с командами обязательно должна начинаться с отступа табуляцией. В качестве имени *правила* обычно выступает имя файла, который получится в результате выполнения *команд* в теле *правила*. *Зависимости* - необязательный список имен файлов, разделенных пробелами, от которых зависит данное *правило*. Если при вызове make окажется, что хотя бы один файл из этого списка новее, чем файл-результат правила, то выполняются все команды из этого правила. В качестве зависимостей могут указываться имена файлов-названия других правил. Тогда make будет выполнять рекурсивную проверку зависимостей. Make не выполняет команды из *правила*, если все файлы-зависимости старее файла-результата.

Пример:

example4a.exe: example4a.rc example4a.cpp myicon.ico

brcc32 example4a.rc

bcc32 -c -tW example4a.cpp

ilink32 -Gn -x -aa -Tpe -Lc:\bcc55\lib c0w32.obj+example4a.obj,example4a.exe,,import32.lib+cw32.lib,,example4a.res

Если не указывать в качестве файлов-зависимостей example4a.rc и example4a.cpp, то make не станет ничего делать, когда файл example4a.exe уже существует. Тем не менее, приведенный пример - не совсем удачный сценарий компиляции. Если мы изменим только файл ресурсов, make все равно будет перекомпилировать исходный текст. Если мы изменим только исходный текст, make будет перекомпилировать еще и ресурсы. С учетом этого замечания, более удачным будет следующий сценарий:

example4a.exe: example4a.obj example4a.res

ilink32 -Gn -x -aa -Tpe -Lc:\bcc55\lib c0w32.obj+example4a.obj,example4a.exe,,import32.lib+cw32.lib,,example4a.res

example4a.obj: example4a.cpp

bcc32 -c -tW example4a.cpp

example4a.res: example4a.rc myicon.ico

brcc32 example4a.rc

Если в командной строке make не указано иное, то make пытается выполнить первое правило из сценария. Именно поэтому первым правилом стоит example4a.exe - результат, который мы хотим получить после компиляции всего проекта.

Если написать подходящий сценарий компиляции, то для компиляции вашего проекта придется набирать лишь команду:

make

1. **Диалоговые окна**

Большинство приложений использует диалоговые окна для запроса у пользователя дополнительной информации для выполнения каких-либо команд. Например, команда открытия файла требует указания имени файла, так что приложение создает диалоговое окно, чтобы запросить у пользователя имя файла. Пока пользователь не укажет имя файла, команда не будет выполнена. После этого программа уничтожает это диалоговое окно. В этом случае используется *модальное* диалоговое окно. Другой пример: текстовый редактор может использовать *немодальное* диалоговое окно, для команды поиска. Пока редактор ищет введенную фразу, диалоговое окно остается на экране. Более того, пользователь может вернуться к редактированию текста, не закрывая диалог поиска. Либо пользователь может ввести другую фразу для поиска. Такое диалоговое окно остается открытым, пока приложение не завершится или пользователь непосредственно не выберет команду закрытия этого диалога.

Чтобы создать диалоговое окно, приложение должно предоставить системе шаблон диалога, описывающий содержание и стиль диалога, и диалоговую процедуру. Диалоговая процедура выполняет примерно такие же задачи, что и процедура обработки событий окна. Диалоговые окна принадлежат к предопределенному классу окон. Windows использует этот класс и соответствующую процедуру обработки событий для модальных и немодальных диалогов. Эта процедура обрабатывает одни сообщения самостоятельно, а другие передает на обработку диалоговой процедуре приложения. У приложения нет непосредственного доступа к этому предопределенному классу и соответствующей ему процедуре обработки событий. Для изменения стиля и поведения диалога программа должна использовать шаблон диалогового окна и диалоговую процедуру.

Для создания модального диалога используется функция **DialogBox**, а для создания немодального диалога - **CreateDialog**:

int WINAPI **DialogBox**(HANDLE hInst, LPCSTR template, HWND parent, DLGPROC DlgFunc)

HWND WINAPI **CreateDialog**(HANDLE hInst, LPCSTR template, HWND parent, DLGPROC DlgFunc)

Параметры: hInst - дескриптор экземпляра программы (модуля, в котором находится шаблон); template - имя ресурса, описывающего диалог; parent - дескриптор родительского окна; DlgFunc - диалоговая функция следующего формата:

BOOL CALLBACK *DlgFunc*(HWND hw, UINT msg, WPARAM wp, LPARAM lp)

Параметры диалоговой функции такие же, как у обычной функции обработки событий. Отличие этой функции - она вызывается из предопределенной функции обработки событий для диалоговых окон. Она должна вернуть значение TRUE, если обработала переданное ей сообщение, или FALSE в противном случае. Она ни в коем случае не должна сама вызывать **DefWindowProc**.

При создании диалогового окна диалоговая процедура получает сообщение WM\_INITDIALOG. Если в ответ на это сообщение процедура возвращает FALSE, диалог не будет создан: функция **DialogBox** вернет значение -1, а **CreateDialog** - NULL.

Модальное диалоговое окно блокирует указанное в качестве родительского окно и появляется поверх него (вне зависимости от стиля WS\_VISIBLE). Приложение закрывает модальное диалоговое окно при помощи функции

BOOL WINAPI **EndDialog**(HWND hw, int result)

Приложение должно вызвать эту функцию из диалоговой процедуры в ответ на сообщение от кнопок "OK", "Cancel" или команды "Close" из системного меню диалога. Параметр result передается программе как результат возврата из функции **DialogBox**.

Немодальное диалоговое окно появляется поверх указанного в качестве родительского окна, но не блокирует его. Диалоговое окно остается поверх родительского окна, даже если оно неактивно. Программа сама отвечает за отображение/сокрытие окна (с помощью стиля WS\_VISIBLE и функции **ShowWindow**). Сообщения для немодального диалогового окна оказываются в основной очереди сообщений программы. Чтобы эти сообщения были корректно обработаны, следует включить в цикл обработки сообщений вызов функции:

BOOL WINAPI **IsDialogMessage**(HWND hwDlg, MSG \*lpMsg)

Если эта функция вернула TRUE, то сообщение обработано и его не следует передавать функциям **TranslateMessage** и **DispatchMessage**.

Немодальное диалоговое окно уничтожается, если уничтожается его родительское окно. Во всех остальных случаях программа должна сама заботиться об уничтожении немодального диалогового окна, используя вызов:

BOOL WINAPI **DestroyWindow**(HWND hw)

Шаблон диалогового окна в ресурсах задается следующим образом:

nameID **DIALOG** [load-option] [mem-option] x, y, width, height

[*property-statements*]

BEGIN

*control-statements*

...

END

В начале блока описания диалога задается: nameID - целочисленный или строковой идентификатор ресурса, x, y - координаты диалога на экране (или относительно родительского окна), width, height - размер диалога. Координаты и размеры диалога и всех элементов внутри него задаются в *диалоговых единицах* (*dialog units*). Одна горизонтальная диалоговая единица соответствует 1/4 средней ширины символа в системном шрифте. Одна вертикальная диалоговая единица соответствует 1/8 средней высоты символа в системном шрифте.

После заголовка блока идет ряд необязательных операторов-параметров диалога (*property-statements*) в любом порядке:

**STYLE** style /\* стиль диалога \*/

**CAPTION** captiontext /\* заголовок диалога \*/

**FONT** pointsize, typeface /\* шрифт диалога (размер, название) \*/

**MENU** menuname /\* меню диалога \*/

В качестве стиля диалога можно применять все перечисленные для обычных окон стили. Обычно выбирают: WS\_POPUP, WS\_SYSMENU, WS\_CAPTION, а также WS\_BORDER для немодального диалога и DS\_MODALFRAME - для модального. Кроме того, можно использовать DS\_SETFOREGROUND, чтобы при отображении диалога перевести его на передний план, даже если его родительское окно неактивно.

В теле шаблона (*control-statements*) перечисляются составляющие его управляющие элементы. Один из возможных вариантов оператора:

**CONTROL** text, id, class, style, x, y, width, height

Здесь text - текст управляющего элемента (заголовок), id - целочисленный идентификатор элемента 0...65535 (внутри одного диалога идентификаторы всех элементов должны различаться), class - имя класса, к которому принадлежит управляющий элемент, style - стиль управляющего элемента, x, y, width, height - положение и размер диалогового элемента относительно клиентской области диалога в диалоговых единицах.

Вот пример диалога, содержащего простое статическое текстовое поле и кнопку "OK":

Ex4\_Dlg **DIALOG** 50,50,90,40

**STYLE** WS\_POPUP|WS\_CAPTION|DS\_MODALFRAME

**CAPTION** "MyDlg"

**FONT** 10, "Arial"

{

**CONTROL** "", 1, "STATIC", SS\_LEFT, 5, 5, 80, 10

**CONTROL** "OK", 2, "BUTTON", BS\_DEFPUSHBUTTON, 5, 20, 80, 12

}

Добавим этот диалог к [ресурсам примера 4](http://dims.karelia.ru/win32/#ex4res). В текст программы добавим две глобальных переменных:

char buf[256]=""; /\* строка для текстового поля в диалоге \*/

HINSTANCE h; /\* дескриптор экземпляра программы \*/

Присвоим переменной h значение дескриптора экземпляра программы в самом начале функции WinMain. Это значение нам потребуется для вызова функции **DialogBox**.

Изменим обработчик сообщения WM\_COMMAND следующим образом:

case WM\_COMMAND:

switch (LOWORD(wp)) {

case 6: /\* *команда меню Exit* \*/

PostQuitMessage(0);

default: /\* *все остальные команды* \*/

wsprintf(buf,"Command code: %d",LOWORD(wp));

DialogBox(h,"Ex4\_Dlg",hw,DlgProc);

}

return 0;

Теперь в текст программы необходимо добавить диалоговую процедуру:

BOOL CALLBACK DlgProc(HWND hw, UINT msg, WPARAM wp, LPARAM lp) {

switch (msg) {

case WM\_INITDIALOG: /\* *сообщение о создании диалога* \*/

SetDlgItemText(hw,1,buf);

return TRUE;

case WM\_COMMAND: /\* *сообщение от управляющих элементов* \*/

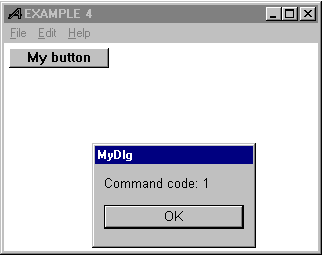
if (LOWORD(wp)==2) EndDialog(hw,0);

}

return FALSE;

}

При создании диалога вызывается процедура **SetDlgItemText**, меняющая содержание текстового поля в диалоге (элемент с id=1). Для уничтожения диалога используется кнопка "OK", генерирующая сообщение WM\_COMMAND с id=2.

Функция DlgProc должна быть определена или описана до ссылки на нее в вызове **DialogBox**.  


1. **Управляющие элементы**

Управляющие элементы, как и другие окна, принадлежат тому или иному классу окон. Windows предоставляет несколько предопределенных классов управляющих элементов. Программа может создавать управляющие элементы поштучно при помощи функции **CreateWindow** или оптом, загружая их вместе с шаблоном диалога из своих ресурсов. Управляющие элементы - это всегда дочерние окна. Управляющие элементы при возникновении некоторых событий, связанных с реакцией пользователя, посылают своему родительскому окну *сообщения-оповещения* (*notification messages*) WM\_COMMAND или WM\_NOTIFY.

Как и любое другое окно, управляющий элемент может быть скрыт или отображен при помощи функции **ShowWindow**. Аналогично, управляющий элемент может быть блокирован или разблокирован при помощи функции:

BOOL WINAPI **EnableWindow**(HWND hw,BOOL bEnable)

В качестве второго параметра передается флаг TRUE (разблокировать) или FALSE (блокировать). Функция возвращает значение TRUE, если перед ее вызовом окно было заблокировано. Узнать текущий статус блокирования окна можно при помощи функции:

BOOL WINAPI **IsWindowEnabled**(HWND hw),

которая возвращает значение TRUE, если окно разблокировано.

Для многих управляющих элементов определены специальные сообщения, которые управляют видом или поведением таких элементов или позволяют получить параметры их состояния. Как и для любого другого окна эти сообщения можно отправить с помощью функции:

LRESULT WINAPI **SendMessage**(HWND hw, UINT msg, WPARAM wp, LPARAM lp)

Все упомянутые функции работают с дескриптором окна, который для управляющих элементов в случае создания диалога по шаблону из ресурсов непосредственно неизвестен, но может быть получен по дескриптору диалога и идентификатору управляющего элемента вызовом:

HWND WINAPI **GetDlgItem**(HWND hDlg, int idDlgItem)

Для функции отсылки сообщений есть специальный вариант, предназначенный для более удобной работы с управляющими элементами:

LRESULT WINAPI **SendDlgItemMessage**(

HWND hwndDlg, /\* дескриптор родительского диалога \*/

int idControl, /\* идентификатор управляющего элемента \*/

UINT msg, /\* код сообщения \*/

WPARAM wp, /\* параметр сообщения \*/

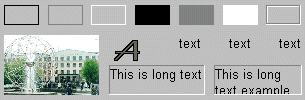
LPARAM lp /\* параметр сообщения \*/

)

Для управляющих элементов внутри диалогов специальный смысл имеют стили WS\_TABSTOP и WS\_GROUP. Если в диалоге имеются управляющие элементы со стилем WS\_TABSTOP, то при нажатии пользователем на клавишу [Tab] (или [Shift]+[Tab]), текущий активный элемент диалога будет терять фокус и передавать его следующему за ним (или предыдущему) ближайшему элементу со стилем WS\_TABSTOP. С помощью стиля WS\_GROUP элементы диалога можно объединять в группы. Группа элементов начинается с элемента со стилем WS\_GROUP и заканчивается элементом, после которого идет элемент со стилем WS\_GROUP, или последним элементом в диалоге. Внутри группы только первый элемент должен иметь стиль WS\_GROUP. Windows допускает перемещение внутри группы при помощи клавиш-стрелок.

Классы предопределенных управляющих элементов:

**"STATIC"**

*Статический управляющий элемент* представляет собой текстовую метку или прямоугольник. Не предоставляет пользователю ни средств ввода, ни вывода. Примеры:  
  
Соответствующее описание ресурсов:

**CONTROL** "",-1, "STATIC", SS\_BLACKFRAME, 5, 40, 20, 10

**CONTROL** "",-1, "STATIC", SS\_GRAYFRAME, 30, 40, 20, 10

**CONTROL** "",-1, "STATIC", SS\_WHITEFRAME, 55, 40, 20, 10

**CONTROL** "",-1, "STATIC", SS\_BLACKRECT, 80, 40, 20, 10

**CONTROL** "",-1, "STATIC", SS\_GRAYRECT, 105, 40, 20, 10

**CONTROL** "",-1, "STATIC", SS\_WHITERECT, 130, 40, 20, 10

**CONTROL** "",-1, "STATIC", SS\_ETCHEDFRAME,155, 40, 20, 10

/\* Для статиков-иконок или картинок текстовое поле определяет имя ресурса \*/

**CONTROL** "Ex4\_Bmp",-1, "STATIC", SS\_BITMAP, 5, 55, -1, -1

**CONTROL** "Ex4\_Icon",-1, "STATIC", SS\_ICON, 65, 55, -1, -1

**CONTROL** "text",-1, "STATIC", SS\_LEFT, 105, 55, 20, 10

**CONTROL** "text",-1, "STATIC", SS\_CENTER, 130, 55, 20, 10

**CONTROL** "text",-1, "STATIC", SS\_RIGHT, 155, 55, 20, 10

/\* По умолчанию SS\_LEFT, SS\_RIGHT, SS\_CENTER делают перенос по словам \*/

**CONTROL** "This is long text example",-1, "STATIC", SS\_SIMPLE|SS\_SUNKEN, 65, 70, 55, 15

**CONTROL** "This is long text example",-1, "STATIC", SS\_LEFT|SS\_SUNKEN, 125, 70, 50, 15

Для текстовых статиков со стилями SS\_LEFT, SS\_RIGHT или SS\_CENTER существуют более простые операторы объявления ресурсов:

**LTEXT** "text",-1, 105, 55, 20, 10

**CTEXT** "text",-1, 130, 55, 20, 10

**RTEXT** "text",-1, 155, 55, 20, 10

**LTEXT** "This is long text example",-1, 65, 70, 55, 15, SS\_LEFTNOWORDWRAP|SS\_SUNKEN

**LTEXT** "This is long text example",-1,125, 70, 50, 15, SS\_LEFT|SS\_SUNKEN

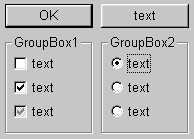
Чтобы поменять текст статика ему можно послать сообщение WM\_SETTEXT (wp=0; lp=(LPARAM)(LPCSTR)lpsz - адрес строки) или использовать функции:

BOOL WINAPI **SetWindowText**(HWND hw, LPCSTR lpsz)

BOOL WINAPI **SetDlgItemText**(HWND hDlg, int idControl, LPCTSTR lpsz)

Чтобы сменить иконку или картинку нетекстового статика, надо послать ему сообщение STM\_SETIMAGE (wp=(WPARAM)fImageType - тип изображения: IMAGE\_BITMAP или IMAGE\_ICON; lp=(LPARAM)(HANDLE)hImage - дескриптор иконки или картинки).

**"BUTTON"**

*Кнопка* - это небольшое прямоугольное дочернее окно, обычно имеющее два состояния: нажато/отпущено или включено/выключено. Пользователь меняет состояние этого элемента щелчком мыши. К этому классу относятся: кнопки-"давилки" (*push buttons*), кнопки-"галочки" (*check boxes*), "радио"-кнопки (*radio buttons*) и специальный тип групповых рамочек (*group boxes*). Примеры:  
  
Соответствующее описание ресурсов:

/\* DEFPUSHBUTTON - кнопка по умолчанию (нажимается по [Enter]) \*/

**CONTROL** "OK", 2, "BUTTON", BS\_DEFPUSHBUTTON, 5, 20, 50, 12

**CONTROL** "text", 3, "BUTTON", BS\_PUSHBUTTON, 60, 20, 50, 12

**CONTROL** "GroupBox1", -1, "BUTTON", BS\_GROUPBOX, 5, 35, 50, 50

**CONTROL** "text", 4, "BUTTON", BS\_CHECKBOX, 10, 45, 30, 10

**CONTROL** "text", 5, "BUTTON", BS\_AUTOCHECKBOX, 10, 57, 30, 10

**CONTROL** "text", 6, "BUTTON", BS\_AUTO3STATE, 10, 69, 30, 10

**CONTROL** "GroupBox2", -1, "BUTTON", BS\_GROUPBOX, 60, 35, 50, 50

**CONTROL** "text", 7, "BUTTON", BS\_AUTORADIOBUTTON|WS\_GROUP, 65, 45, 30, 10

**CONTROL** "text", 8, "BUTTON", BS\_AUTORADIOBUTTON, 65, 57, 30, 10

**CONTROL** "text", 9, "BUTTON", BS\_AUTORADIOBUTTON, 65, 69, 30, 10

Для кнопок существуют более простые операторы объявления ресурсов:

**DEFPUSHBUTTON** text, id, x, y, width, height [, style]

**PUSHBUTTON** text, id, x, y, width, height [, style]

**GROUPBOX** text, id, x, y, width, height [, style]

**CHECKBOX** text, id, x, y, width, height [, style]

**RADIOBUTTON** text, id, x, y, width, height [, style]

Разница между стилями BS\_*xxx* и BS\_AUTO*xxx* заключается в том, что при щелчке по AUTO-кнопкам Windows сама автоматически переключает их состояние. Для не AUTO-кнопок это надо делать вручную в диалоговой процедуре, послав сообщение BM\_SETCHECK (wp=(WPARAM)fCheck - флаг: BST\_UNCHECKED, BST\_CHECKED или BST\_INDETERMINATE (для BS\_3STATE-кнопок); lp=0) или при помощи функций:

BOOL WINAPI **CheckDlgButton**(HWND hDlg, int idButton, UINT fCheck)

BOOL WINAPI **CheckRadioButton**(

HWND hDlg, /\* дескриптор родительского диалога \*/

int idFirstButton, /\* id первой радио-кнопки в группе \*/

int idLastButton, /\* id последней радио-кнопки в группе \*/

int idCheckButton /\* id отмечаемой радио-кнопки \*/

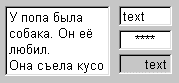
)

Автоматические радио-кнопки должны быть объединены в группу при помощи стиля WS\_GROUP, чтобы Windows корректно их обрабатывала.  
Проверить состояние кнопки можно, послав ей сообщение BM\_GETCHECK (wp=0; lp=0) или вызовом функции:

UINT WINAPI **IsDlgButtonChecked**(HWND hDlg, int idButton)

При щелчке мыши по кнопке она присылает родительскому диалогу сообщение-оповещение WM\_COMMAND (HIWORD(wp)=BN\_CLICKED; LOWORD(wp)=(int)idButton; lp=(HWND)hwndButton).

**"EDIT"**

*Поле редактирования* предназначено для ввода пользователем текста с клавиатуры. Щелчком мыши внутри элемента пользователь передает этому элементу *фокус ввода* (*input focus*). При этом внутри элемента появляется текстовый курсор - мигающая каретка. Пользователь может использовать мышь для перемещения каретки по полю редактирования и выделению текста в этом поле. Примеры:  
  
Соответствующее описание ресурсов:

/\* По умолчанию эти элементы создаются вообще без рамки, поэтому добавлено WS\_BORDER \*/

**CONTROL** "" 4, "EDIT", ES\_MULTILINE|ES\_WANTRETURN|WS\_BORDER 5, 45, 60, 35

**CONTROL** "text", 5, "EDIT", ES\_LEFT|WS\_BORDER, 70, 45, 30, 10

**CONTROL** "text", 6, "EDIT", ES\_CENTER|ES\_PASSWORD|WS\_BORDER, 70, 57, 30, 10

**CONTROL** "text", 7, "EDIT", ES\_RIGHT|ES\_READONLY|WS\_BORDER, 70, 69, 30, 10

Стиль ES\_WANTRETURN означает, что кнопка [Enter] будет обрабатываться самим элементом, а не передаваться диалогу. Благодаря этому стилю оказался возможен переход на новую строчку для предложения "Она съела кусок..." (на картинке).  
По умолчанию текстовые поля позволяют вводить столько текста, сколько может отобразиться в рамках поля. Чтобы предоставить пользователю возможность ввести больше текста, надо использовать стиль ES\_AUTOHSCROLL (и ES\_AUTOVSCROLL для многострочных полей).  
Для текстовых полей существует более простой оператор объявления ресурсов:

**EDITTEXT** id, x, y, width, height [, style]

Чтобы поменять содержимое текстового поля, программа вызывает функцию **SetDlgItemText**. Чтобы получить текущее содержимое текстового поля, используется функция:

UINT WINAPI **GetDlgItemText**(

HWND hDlg, /\* дескриптор родительского диалога \*/

int idControl, /\* идентификатор поля \*/

LPSTR lpString, /\* буфер под текст \*/

int nMaxCount /\* размер буфера \*/

)

Чтобы узнать размер строки в текстовом поле, надо послать элементу сообщение WM\_GETTEXTLENGTH (wp=0; lp=0).  
Текстовое поле посылает родительскому диалогу следующие сообщения-оповещения WM\_COMMAND (LOWORD(wp)=(int)idContol; lp=(HWND)hwndEditCtrl):

* + HIWORD(wp)=EN\_KILLFOCUS - текстовое поле потеряло фокус (фокус передан другому элементу диалога);
  + HIWORD(wp)=EN\_SETFOCUS - текстовое поле получило фокус;
  + HIWORD(wp)=EN\_CHANGE - пользователь изменил текст в поле;
  + HIWORD(wp)=EN\_ERRSPACE - закончилось место, отведенное под текстовый буфер управляющего элемента.

**"LISTBOX"**

*Окно-список* используется для отображения списка имен (например, имен файлов). Пользователь может, просматривая список, выделить один или несколько элементов щелчком мыши. При выделении того или иного элемента списка, он подсвечивается, а родительскому окну посылается сообщение-оповещение. Для очень больших списков могут использоваться полосы прокрутки. Примеры:  
  
Соответствующее описание ресурсов:

**CONTROL** "" 3, "LISTBOX", LBS\_MULTIPLESEL|WS\_BORDER|WS\_VSCROLL, 5, 45, 60, 35

**CONTROL** "" 4, "LISTBOX", LBS\_MULTICOLUMN|WS\_BORDER|WS\_HSCROLL, 70, 45, 60, 35

**CONTROL** "" 5, "LISTBOX", LBS\_SORT|LBS\_NOSEL|WS\_BORDER, 135, 45, 60, 35

или в короткой форме:

**LISTBOX** 3, 5, 45, 60, 35, LBS\_MULTIPLESEL|WS\_BORDER|WS\_VSCROLL

**LISTBOX** 4, 70, 45, 60, 35, LBS\_MULTICOLUMN|WS\_BORDER|WS\_HSCROLL

**LISTBOX** 5, 135, 45, 60, 35, LBS\_SORT|LBS\_NOSEL|WS\_BORDER

Для добавления элемента к списку следует послать ему сообщение LB\_ADDSTRING (wp=0; lp=(LPARAM)(LPCSTR)lpsz - строка для добавления). Для того, чтобы заполнить один из списков, показанных на рисунке, в обработчик сообщения WM\_INITDIALOG в диалоговую процедуру был вставлен такой фрагмент:

char \*a[12]={

"jan","feb","mar","apr","may","jun",

"jul","aug","sep","oct","nov","dec"};

for (int i=0; i<12; i++) {

SendDlgItemMessage(hw,3,LB\_ADDSTRING,0,(LPARAM)a[i]);

}

Кроме этого, список "понимает" следующие сообщения:

* + LB\_DELETESTRING (wp=(WPARAM)index; lp=0) - удалить элемент с указанным номером;
  + LB\_INSERTSTRING (wp=(WPARAM)index; lp=(LPARAM)(LPCSTR)lpsz) - вставить указанную строку в список как элемент с индексом *index*;
  + LB\_FINDSTRING (wp=(WPARAM)indexStart; lp=(LPARAM)(LPTSTR)lpszFind) - найти элемент, содержащий указанную строку (поиск ведется, начиная с элемента *indexStart*), результат сообщения - номер элемента, удовлетворяющего критерию, или LB\_ERR;
  + LB\_GETCOUNT (wp=0; lp=0) - количество элементов в списке;
  + LB\_GETCURSEL (wp=0; lp=0) - выделенный элемент в списке;
  + LB\_RESETCONTENT (wp=0; lp=0) - удалить все элементы из списка.

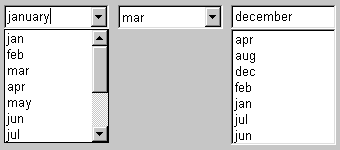
Окно-список посылает родительскому диалогу следующие сообщения-оповещения WM\_COMMAND (LOWORD(wp)=(int)idContol; lp=(HWND)hwndListBox):

* + HIWORD(wp)=LBN\_DBLCLK - пользователь дважды щелкнул мышью по списку;
  + HIWORD(wp)=LBN\_SELCHANGE - пользователь выделил другой элемент в списке (или отменил выделение).

**"COMBOBOX"**

*Комбобокс* - это помесь поля редактирования с окном-списком. Этот элемент содержит поле редактирование и список, который может отображаться все время либо "выпадать" при нажатии на кнопку рядом с полем редактирования. Есть три основных типа комбобоксов:

* + *"выпадающий" комбобокс* (CBS\_DROPDOWN) содержит поле редактирования и "выпадающий" список;
  + *"выпадающий" список* (CBS\_DROPDOWNLIST) не содержит поля для изменения текста;
  + *простой комбобокс* (CBS\_SIMPLE) содержит поле редактирования и обычный список.

  
Обратите внимание, что в ресурсах значение высоты элемента определяет размер поля редактирования вместе с "выпавшим" списком по вертикали.

**CONTROL** "" 3, "COMBOBOX", CBS\_DROPDOWN|CBS\_AUTOHSCROLL|WS\_VSCROLL, 5, 45, 60, 70

**CONTROL** "" 4, "COMBOBOX", CBS\_DROPDOWNLIST|WS\_VSCROLL, 70, 45, 60, 70

**CONTROL** "" 5, "COMBOBOX", CBS\_SIMPLE|CBS\_SORT, 135, 45, 60, 70

Короткий вариант этого же объявления ресурсов:

**COMBOBOX** 3, 5, 45, 60, 70, CBS\_DROPDOWN|CBS\_AUTOHSCROLL|WS\_VSCROLL

**COMBOBOX** 4, 70, 45, 60, 70, CBS\_DROPDOWNLIST|WS\_VSCROLL

**COMBOBOX** 5, 135, 45, 60, 70, CBS\_SIMPLE|CBS\_SORT

Для работы с комбобоксами существуют сообщения, аналогичные списковым: CB\_ADDSTRING, CB\_DELETESTRING, CB\_INSERTSTRING, CB\_FINDSTRING, CB\_GETCOUNT, CB\_GETCURSEL, CB\_RESETCONTENT.  
Комбобокс посылает родительскому диалогу сообщение оповещение WM\_COMMAND со следующими кодами оповещения:

* + CBN\_SELCHANGE, когда пользователь выделяет другую строку в комбобоксе (бывает полезным для простых комбобоксов);
  + CBN\_SELENDOK, когда пользователь выбрал элемент в выпадающем списке и щелкнул мышкой по выделению (подтвердил выделение), для простых комбобоксов посылается перед каждым CBN\_SELCHANGE;
  + CBN\_SELENDCANCEL, когда пользователь закрыл выпадающий список, так и не выбрав никакой элемент;
  + CBN\_DROPDOWN, когда открывается выпадающий список;
  + CBN\_CLOSEUP, когда выпадающий список был закрыт по той или иной причине.

Кроме предопределенных управляющих элементов, Windows предоставляет еще набор стандартных управляющих элементов посредством библиотеки Common Controls Library (COMCTL32.DLL). Чтобы воспользоваться ей, в тест программы надо включить заголовочный файл commctrl.h и добавить в блок инициализации программы вызов функции:

void WINAPI **InitCommonControls**(void)

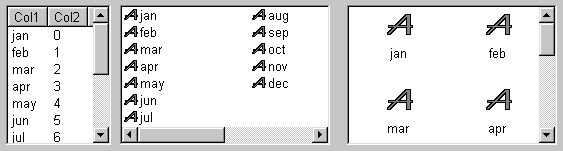
Управляющие элементы из этой библиотеки, как правило, посылают сообщения-оповещения родительскому диалогу через сообщение WM\_NOTIFY (wp=(int)idControl; lp=(LPARAM)(NMHDR\*)pmnh - указатель на структуру со специльными параметрами сообщения-оповещения).

Классы управляющих элементов из Common Controls Library:

**List View Controls (WC\_LISTVIEW)**

*Элемент просмотра списков* - это окно отображающее совокупность элементов. Каждый элемент может быть представлен текстовой меткой и (необязательно) иконкой. Типичный пример использования этого элемента - программа "Проводник". Содержимое того или иного каталога представляется в виде элемента просмотра списков. Есть четыре основных стиля для этого элемента:

* + крупные иконки - стиль LVS\_ICON;
  + мелкие иконки - стиль LVS\_SMALLICON;
  + список - стиль LVS\_LIST;
  + таблица - стиль LVS\_REPORT.



**CONTROL** "",3,WC\_LISTVIEW,LVS\_REPORT|WS\_BORDER, 5, 45, 60, 70

**CONTROL** "",4,WC\_LISTVIEW,LVS\_LIST|WS\_BORDER, 70, 45, 120, 70

**CONTROL** "",5,WC\_LISTVIEW,LVS\_ICON|WS\_BORDER, 200, 45, 120, 70

Приведенные в примере списки заполнялись в диалоговой процедуре при инициализации диалога (WM\_INITDIALOG):

/\* Создание колонок для 1го списка \*/

LV\_COLUMN lc; lc.mask=LVCF\_FMT|LVCF\_TEXT|LVCF\_SUBITEM|LVCF\_WIDTH;

lc.fmt=LVCFMT\_LEFT;

lc.pszText="Col1"; lc.iSubItem=0; lc.cx=40;

SendDlgItemMessage(hw,3,LVM\_INSERTCOLUMN,0,(LPARAM)&lc);

lc.pszText="Col2"; lc.iSubItem=1; lc.cx=40;

SendDlgItemMessage(hw,3,LVM\_INSERTCOLUMN,1,(LPARAM)&lc);

/\* Создание списка иконок для 2го и 3го списков \*/

HIMAGELIST himl1,himl2;

himl1=ImageList\_Create(16,16,ILC\_MASK,1,0); /\* список маленьких иконок \*/

ImageList\_AddIcon(himl1,LoadIcon(h,"Ex4\_Icon"));

himl2=ImageList\_Create(32,32,ILC\_MASK,1,0); /\* список больших иконок \*/

ImageList\_AddIcon(himl2,LoadIcon(h,"Ex4\_Icon"));

SendDlgItemMessage(hw,4,LVM\_SETIMAGELIST,LVSIL\_SMALL,(LPARAM)himl1);

SendDlgItemMessage(hw,5,LVM\_SETIMAGELIST,LVSIL\_NORMAL,(LPARAM)himl2);

/\* Заполнение списков \*/

LV\_ITEM li; li.mask=LVIF\_TEXT|LVIF\_IMAGE;

li.iImage=0; /\* номер иконки в списке \*/

for (int i=0; i<12; i++) {

li.iItem=i;

li.iSubItem=0; li.pszText=a[i];

SendDlgItemMessage(hw,3,LVM\_INSERTITEM,0,(LPARAM)&li);

SendDlgItemMessage(hw,4,LVM\_INSERTITEM,0,(LPARAM)&li);

SendDlgItemMessage(hw,5,LVM\_INSERTITEM,0,(LPARAM)&li);

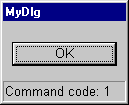
wsprintf(str,"%d",i); /\* вторая колонка для 1го списка \*/

li.iSubItem=1; li.pszText=str;

SendDlgItemMessage(hw,3,LVM\_SETITEM,0,(LPARAM)&li);

}

**Status Windows (STATUSCLASSNAME)**

*Поле статуса* - это горизонтальное окно в нижней части родительского окна, которое программа обычно использует для отображения каких-либо характеристик, параметров или небольших текстовых сообщений.  
  
В [примере 4б](http://dims.karelia.ru/win32/#ex4b) можно заменить статическое текстовое поле на поле статуса:

Ex4\_Dlg **DIALOG** 50,50,70,40

**STYLE** WS\_POPUP|WS\_CAPTION|WS\_BORDER

**CAPTION** "MyDlg"

**FONT** 10, "Arial"

{

**CONTROL** "OK", 2, "BUTTON", BS\_DEFPUSHBUTTON, 5, 10, 60, 12

**CONTROL** "Status text",1,STATUSCLASSNAME, 0, 0, 0, 0, 0

}

При создании поля статуса не требуется указывать ни размер, ни позицию окна, Windows сама выберет эти параметры подходящим образом. Для создания поля статуса можно использовать специальную функцию:

HWND WINAPI **CreateStatusWindow**(

LONG style, /\* стиль: обязательно указываем WS\_CHILD|WS\_VISIBLE \*/

LPCTSTR lpszText, /\* текст поля статуса \*/

HWND hwndParent, /\* родительское окно \*/

UINT wID /\* числовой id поля статуса \*/

)

С помощью сообщения SB\_SETPARTS поле статуса можно разбить на части (wp=(int)nParts - количество частей; lp=(LPARAM)(int\*)widths - указатель на массив размеров частей). В таком случае текст для каждой части поля статуса задается сообщением SB\_SETTEXT (wp=(int)iPart - номер части; lp=(LPARAM)(LPSTR)lpszText - текстовая строка). Пример:

CreateStatusWindow(WS\_CHILD|WS\_VISIBLE|SBARS\_SIZEGRIP,"text",hw,100);

int widths[5]={100,150,200,-1};

SendDlgItemMessage(hw,100,SB\_SETPARTS,4,(LPARAM)widths);

SendDlgItemMessage(hw,100,SB\_SETTEXT,2,(LPARAM)"part 2");

SendDlgItemMessage(hw,100,SB\_SETTEXT,3,(LPARAM)"last part");

http://dims.karelia.ru/win32/example4c.gif

**Up-Down Controls (UPDOWN\_CLASS)**

Управляющий элемент *"up-down"* представляет собой пару небольших кнопок-стрелок, нажимая которые, пользователь увеличивает или уменьшает значение. Этот элемент, как правило, связывается с *элементом-компаньоном* (*buddy window*), обычно реализованным в виде поля редактирования. Для пользователя элемент "up-down" и его компаньон представляются единым управляющим элементом.  
http://dims.karelia.ru/win32/updown.gif

**CONTROL** "0", 5, "EDIT", ES\_LEFT|WS\_BORDER, 5, 30, 30, 10

**CONTROL** "", 6, UPDOWN\_CLASS, UDS\_AUTOBUDDY|UDS\_SETBUDDYINT, 35, 30, 10, 10

Если при создании элемента "up-down" указать стиль UDS\_AUTOBUDDY, то компаньоном будет назначен предыдущий управляющий элемент диалога. Программа может также передать дескриптор окна-компаньона при помощи сообщения UDM\_SETBUDDY (wp=(WPARAM)(HWND)hwndBuddy - дескриптор окна-компаньона; lp=0). Если элементу "up-down" назначить стиль UDS\_SETBUDDYINT, то он будет автоматически менять текст окна-компаньона, представляющий числовое значение.  
Другой способ создать элемент "up-down" - использовать функцию

HWND WINAPI **CreateUpDownControl**(

DWORD dwStyle, /\* стиль элемента \*/

int x, int y, /\* позиция \*/

int cx, int cy, /\* размеры \*/

HWND hParent, /\* дескриптор родительского окна \*/

int ID, /\* id элемента \*/

HINSTANCE hInst, /\* дескриптор экземпляра программы \*/

HWND hBuddy, /\* дескриптор окна-компаньона \*/

int nUpper, /\* максимальное значение \*/

int nLower, /\* минимальное значение \*/

int nPos /\* текущее значение \*/

)

**Progress Bars (PROGRESS\_CLASS)**

*Полоса прогресса* - это окно, которое программа может использовать для индикации состояния выполнения какой-либо длительной операции. Окно представляет собой прямоугольник, заполняемый системным цветом слева направо.  
http://dims.karelia.ru/win32/progresses.gif

**CONTROL** "",3,PROGRESS\_CLASS,WS\_BORDER, 5, 45, 100, 10

Каждый раз, когда приложение посылает этому окну сообщение PBM\_STEPIT (wp=0; lp=0), заполнение полосы прогресса продвигается дальше вправо на некоторое значение.

**Tooltip Controls (TOOLTIPS\_CLASS)**

*Окно-подсказка* - всплывающее окно, содержащее строку описательной информации о том или ином элементе интерфейса программы. Таким элементом интерфейса может быть конкретное окно (управляющий элемент) или прямоугольный участок клиентской области какого-либо окна. Большую часть времени подсказка скрыта. Она появляется, когда пользователь задерживат курсор мыши над тем или иным элементом интерфейса программы более, чем на полсекунды. Подсказка скрывается, когда пользователь кликает мышью или уводит курсор с этого элемента. Одно окно-подсказка может обслуживать любое количество элементов интерфейса. Чтобы назначить тому или иному элементу интерфейса программы подсказку, надо окну-подсказке послать сообщение TTM\_ADDTOOL (wp=0; lp=(LPARAM)(TOOLINFO\*)lpti - указатель на структуру, содержащую информацию об элементе). Пример:

TOOLINFO ti;

HWND hwTooltip=CreateWindow(TOOLTIPS\_CLASS,"",TTS\_ALWAYSTIP,

CW\_USEDEFAULT, CW\_USEDEFAULT, CW\_USEDEFAULT, CW\_USEDEFAULT,NULL,NULL,h,NULL);

ti.cbSize=sizeof(TOOLINFO);

ti.uFlags=TTF\_SUBCLASS|TTF\_IDISHWND;

ti.hwnd=hMainWnd;

ti.uId=(UINT)GetDlgItem(hMainWnd,ID\_MYBUTTON);

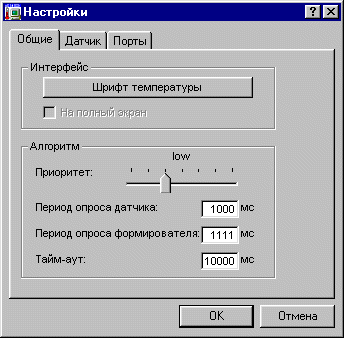
ti.hinst=h;

ti.lpszText="Tooltip for my button";

SendMessage(hwTooltip,TTM\_ADDTOOL,0,(LPARAM)&ti);

http://dims.karelia.ru/win32/tooltip.gif

**Property Sheets & Tab Controls**

Элементы *вкладки свойств* и *переключатели вкладок* обычно используются совместно. Пример использования вкладок:  
  
Каждая вкладка содержит свой набор управляющих элементов и может задаваться в ресурсах так же, как и отдельный диалог:

/\* Шаблон вкладки "Общие" \*/

ODTab\_General DIALOG 0,0,180,150

CAPTION "Общие"

STYLE WS\_CHILD|WS\_VISIBLE

FONT 8,"Arial"

{

GROUPBOX "Интерфейс",-1,5,5,170,40

PUSHBUTTON "Шрифт температуры",IDC\_FONT,20,15,140,12

AUTOCHECKBOX "На полный экран",IDC\_FULLSCR,20,30,140,10,WS\_DISABLED

GROUPBOX "Алгоритм",-1,5,50,170,80

LTEXT "Приоритет:",-1,15,65,50,10

CTEXT "priority",IDC\_PRIOTEXT,70,56,85,8

CONTROL "",IDC\_PRIOTRACK,TRACKBAR\_CLASS,TBS\_TOP|TBS\_AUTOTICKS,70,64,85,17

LTEXT "Период опроса датчика:",-1,15,85,110,10

EDITTEXT IDC\_TRATE,125,85,25,10,ES\_RIGHT

LTEXT "мс",-1,152,85,10,10

LTEXT "Период опроса формирователя:",-1,15,100,110,10

EDITTEXT IDC\_FRATE,125,100,25,10,ES\_RIGHT

LTEXT "мс",-1,152,100,10,10

LTEXT "Тайм-аут:",-1,15,115,110,10

EDITTEXT IDC\_TIMEOUT,125,115,25,10,ES\_RIGHT

LTEXT "мс",-1,152,115,10,10

}

/\* Шаблон вкладки "Датчик" \*/

ODTab\_Sensor DIALOG 0,0,180,150

CAPTION "Датчик"

STYLE WS\_CHILD|WS\_VISIBLE

FONT 8,"Arial"

{

/\* ... \*/

}

/\* Шаблон вкладки "Порты" \*/

ODTab\_Ports DIALOG 0,0,200,150

CAPTION "Порты"

STYLE WS\_CHILD|WS\_VISIBLE

FONT 8,"Arial"

{

/\* ... \*/

}

Для создания диалога с вкладками используется функция **PropertySheet**, перед вызовом которой надо заполнить соответствующие системные структуры:

char \*TabTemplts[NumTabs]={"ODTab\_General","ODTab\_Sensor","ODTab\_Ports"};

PROPSHEETPAGE psp[NumTabs]; /\* заполняется для каждой вкладки \*/

for (i=0; i<NumTabs; i++) {

psp[i].dwSize=sizeof(PROPSHEETPAGE);

psp[i].dwFlags=PSP\_DEFAULT;

psp[i].hInstance=hThisInstance;

psp[i].pszTemplate=TabTemplts[i];

psp[i].pfnDlgProc=(DLGPROC)GlobDlgProc;

psp[i].pfnCallback=NULL;

}

PROPSHEETHEADER psh; /\* описывает весь диалог \*/

psh.dwSize=sizeof(PROPSHEETHEADER);

psh.dwFlags=PSH\_NOAPPLYNOW|PSH\_PROPSHEETPAGE;

psh.hwndParent=hWnd;

psh.hInstance=hThisInstance;

psh.pszCaption="Настройки";

psh.nPages=NumTabs;

psh.nStartPage=0;

psh.ppsp=(LPCPROPSHEETPAGE)&psp;

psh.pfnCallback=NULL;

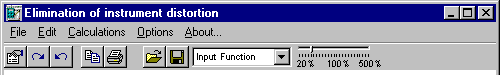
PropertySheet(&psh);

Для каждой вкладки может быть своя диалоговая процедура, а может быть общая для всех вкладок (как в этом примере).

**Trackbars (TRACKBAR\_CLASS)**

*Ползунок* (*бегунок*) используется, если от пользователя требуется получить дискретное значение из определенного диапазона. Маркер ползунка пермещается на заданное программой значение. Пример ползунка показан на первой вкладке предыдущего примера. Ползунки бывают горизонтальные (TBS\_HORZ) или вертикальные (TBS\_VERT). Диапазон значений ползунка задается сообщением TBM\_SETRANGE (wp=(BOOL)fRedraw - перерисовать маркер после изменения диапазона; lp=MAKELONG(lMinimum,lMaximum) - диапазон значений: младшее слово - минимальное значение, старшее слово - максимальное значение). Переместить ползунок можно при помощи сообщения TBM\_SETPOS (wp=TRUE; lp=(LONG)position - новая позиция ползунка). Чтобы получить текущее значение ползунка, следует послать ему сообщение TBM\_GETPOS (wp=0; lp=0). Ползунок оповещает родительское окно о событиях через сообщение WM\_HSCROLL (LOWORD(wp)=ScrollCode - код события; HIWORD(wp)=posistion - позиция маркера ползунка; lp=(HWND)hwndTrackBar - дескриптор элемента).

**Toolbars (TOOLBARCLASSNAME)**

*Панель инструментов* - это окно, содержащее набор кнопок, посылающих командное сообщение родительскому окну, когда пользователь щелкает по ним. Как правило, кнопки на панели инструментов соответствуют часто используемым командам меню приложения. Панель инструментов располагается ниже строки меню.  
  
Информация о кнопках передается панели инструментов через структуру TBBUTTON, а для создания окна панели инструментов удобно использовать функцию **CreateToolbarEx**.

TBBUTTON tbb[nButtons];

/\* 1я кнопка \*/

tbb[0].iBitmap=STD\_PROPERTIES; /\*id иконки\*/

tbb[0].idCommand=ID\_OPTION; /\*id команды\*/

tbb[0].fsState=TBSTATE\_ENABLED; /\*состояние\*/

tbb[0].fsStyle=TBSTYLE\_BUTTON; /\*стиль\*/

tbb[0].iString=0; /\*подпись под кнопкой\*/

/\* 2я кнопка \*/

tbb[1].iBitmap=STD\_REDOW;

tbb[1].idCommand=ID\_CALCOUT;

tbb[1].fsState=TBSTATE\_ENABLED;

tbb[1].fsStyle=TBSTYLE\_BUTTON;

tbb[1].iString=0;

/\* 3я кнопка \*/

tbb[2].iBitmap=STD\_UNDO;

tbb[2].idCommand=ID\_CALCIN;

tbb[2].fsState=TBSTATE\_ENABLED;

tbb[2].fsStyle=TBSTYLE\_BUTTON;

tbb[2].iString=0;

/\* разделитель \*/

tbb[3].fsStyle=TBSTYLE\_SEP;

/\* и т.д. \*/

HWND htb=CreateToolbarEx(

(HWND) hWndMain, /\*дескриптор родительского окна\*/

(DWORD) WS\_CHILD|WS\_VISIBLE|TBSTYLE\_TOOLTIPS, /\*стиль\*/

(UINT) ID\_TOOLBAR, /\*id окна\*/

(int) 15, /\*количество иконок в указанном ресурсе\*/

(HINSTANCE) HINST\_COMMCTRL, /\*дескриптор модуля, из которого берется ресурс с иконками\*/

(UINT) IDB\_STD\_SMALL\_COLOR, /\*id ресурса с иконками\*/

(TBBUTTON\*) tbb, /\*указатель на массив с информацией о кнопках\*/

(int) nButtons, /\*количество кнопок на панели\*/

16,16, /\*размеры кнопок\*/

15,15, /\*размеры иконок для кнопок\*/

sizeof(TBBUTTON) /\*размер структуры TBBUTTON\*/

);

**Rich Edit Controls (RICHEDIT\_CLASS)**

*Продвинутое поле редактирования* является развитием класса "EDIT" стандартных управляющих элементов. Элементы управления этого класса поддерживают форматирование текста (по отдельным символам и по отдельным абзацам) и позволяют внедрять OLE-объекты.

**Tree View Controls (WC\_TREEVIEW)**

*Элемент просмотра дерева* позволяет представлять информацию об иерархии некоторых объектов (содержание документа, дерево каталогов файловой системы и т.п.) Каждый объект может быть представлен текстовой меткой и иконкой. Объект может иметь иерархию дочерних объектов, которая раскрывается по щелчку на этом элементе.

1. **Стандартные диалоги**

Windows предоставляет набор готовых стандартных диалогов посредством библиотеки Common Dialog Boxes Library (COMDLG32.DLL): диалог открытия и сохранения файла, диалог печати документа, диалог выбора цвета, шрифта и т.п. Чтобы создать один из перечисленных диалогов, надо заполнить определенную структуру и вызвать соответствующую функцию из этой библиотеки:

* + BOOL WINAPI **ChooseColor**(CHOOSECOLOR\* lpcc) - создает диалог, отображающий палитру цветов и позволяющий пользователю выбрать тот или иной цвет или создать свой.
  + BOOL WINAPI **ChooseFont**(CHOOSEFONT\* lpcf) - создает диалог, отображающий имена установленных в системе шрифтов, их кегль, стиль начертания и т.п.
  + BOOL WINAPI **GetOpenFileName**(OPENFILENAME\* lpofn) и BOOL WINAPI **GetSaveFileNAme**(OPENFILENAME\* lpofn) - создают диалог, отображающий содержимое того или иного каталога, и позвояющий пользователю выбрать уникальное имя файла для открытия или сохранения.
  + BOOL WINAPI **PrintDlg**(PRINTDLG\* lppd) - создает диалог, позволяющий пользователю установить различные опции печати, например, диапазон страниц, количество копий и др.
  + BOOL WINAPI **PageStupDlg**(PAGESETUPDLG\* lppsd) - создает диалог, позволяющий пользователю выбрать различные параметры страницы: ориентацию, поля, размер бумаги и т.п.
  + HWND WINAPI **FindText**(FINDREPLACE\* lpfr) - создает диалог, позволяющий пользователю ввести строку для поиска и такие опции, как направление поиска.
  + HWND WINAPI **ReplaceText**(FINDREPLACE\* lpfr) - создает диалог, позволяющий пользователю ввести строку для поиска, строку для замены и опции замены (направление поиска, область поиска).

Все перечисленные диалоги, кроме последних двух, - модальные, т.е. указанная функция не вернет значение, пока пользователь тем или иным способом не закроет диалог. Значение TRUE означает, что пользователь закрыл диалог, нажав на "ОК", а соответствующая структура заполнена новыми значениями. Значение FALSE означает, что пользователь нажал на [Esc], выбрал команду системного меню "Закрыть" или нажал кнопку "Отмена", а соответствующая структура осталась неизменной. Пример использования диалога открытия файла:

char filename[MAX\_PATH]=""; /\*буфер под имя файла\*/

OPENFILENAME of;

of.lStructSize=OPENFILENAME\_SIZE\_VERSION\_400A; /\*размер структуры OPENFILENAME\*/

of.hwndOwner=hw; /\*дескриптор родительского окна\*/

of.hInstance=h; /\*дескриптор экземпляра программы\*/

of.lpstrFilter="All files (\*.\*)\0\*.\*\0";/\*фильтр файлов (тип)\*/

of.lpstrCustomFilter=NULL; /\*еще один фильтр: нам не надо\*/

of.nMaxCustFilter=0; /\*нам не надо\*/

of.nFilterIndex=1; /\*количество заданных нами фильтров\*/

of.lpstrFile=filename; /\*адрес буфера под имя файла\*/

of.nMaxFile=MAX\_PATH; /\*размер буфера под имя файла\*/

of.lpstrFileTitle=NULL; /\*буфер под рекомендуемый заголовок: нам не надо\*/

of.nMaxFileTitle=0; /\*нам не надо\*/

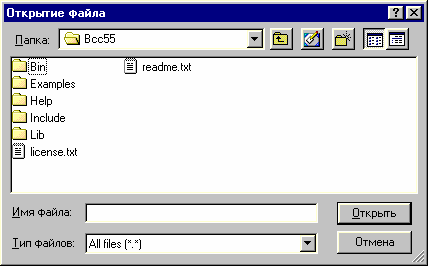
of.lpstrInitialDir=NULL; /\*стартовый каталог: текущий\*/

of.Flags=OFN\_PATHMUSTEXIST|OFN\_FILEMUSTEXIST|OFN\_HIDEREADONLY; /\*разные флаги\*/

if (GetOpenFileName(&of)) {

/\* действия в случае успешного выбора файла \*/

}



1. **Немодальный диалог в качестве главного окна**

Ресурсы значительно облегчают работу программиста по созданию интерфейса программы, поэтому весьма привлекательно использовать ресурсы для описания главного окна. Это возможно в случае, если главное окно реализовано в виде немодального диалога.

Пример 5 демонстрирует использование немодального диалога в приложении типа "блокнот".

Файл **example5.h** содержит константы-идентификаторы команд меню и элементов диалога.

#define IDC\_OPEN 10

#define IDC\_SAVE 11

#define IDC\_SAVEAS 12

#define IDC\_EXIT 13

#define IDC\_ABOUT 14

#define ID\_EDIT 20

#define ID\_STATUS 21

Файл **example5.rc** описывает ресурсы программы: иконку, меню и шаблон диалога.

#include "example5.h"

Ex5\_Icon ICON "myicon.ico"

Ex5\_Menu MENU

{

POPUP "&File"

{

MENUITEM "&Open...", IDC\_OPEN

MENUITEM "&Save", IDC\_SAVE

MENUITEM "Save &As...", IDC\_SAVEAS

MENUITEM SEPARATOR

MENUITEM "&Exit\tAlt-F4", IDC\_EXIT

}

POPUP "&Help"

{

MENUITEM "&About...", IDC\_ABOUT

}

}

Ex5\_Dlg DIALOG 50,50,300,200

STYLE WS\_OVERLAPPED|WS\_CAPTION|WS\_BORDER|WS\_SYSMENU|WS\_VISIBLE

MENU "Ex5\_Menu"

CAPTION "Example 5"

FONT 10, "Arial"

{

EDITTEXT ID\_EDIT, 5, 5, 290, 180, ES\_MULTILINE|ES\_WANTRETURN|ES\_AUTOHSCROLL|ES\_AUTOVSCROLL

CONTROL "", ID\_STATUS, STATUSCLASSNAME, 0, 0, 0, 0, 0

}

Файл **example5.cpp** - текст программы.

#include

#include

#include

#include "example5.h"

BOOL CALLBACK MainProc(HWND,UINT,WPARAM,LPARAM);

HINSTANCE hThisInstance;

char filename[MAX\_PATH]=""; /\*буфер имени файла\*/

int WINAPI WinMain(HINSTANCE hInst,HINSTANCE,LPSTR,int) {

hThisInstance=hInst;

InitCommonControls();

HWND hMainWnd=CreateDialog(hInst,"Ex5\_Dlg",NULL,(DLGPROC)MainProc);

if (!hMainWnd) return FALSE;

MSG msg; /\*цикл обработки событий\*/

while (GetMessage(&msg,NULL,0,0)) {

if (!IsDialogMessage(hMainWnd,&msg)) {

TranslateMessage(&msg);

DispatchMessage(&msg);

}

}

return msg.wParam;

}

/\* диалоговая процедура \*/

BOOL CALLBACK MainProc(HWND hw,UINT msg,WPARAM wp,LPARAM lp) {

static DWORD OldIcon=0; /\* id старой иконки диалога \*/

static OPENFILENAME of;

char\* buf;

HANDLE hf;

DWORD len,len1;

switch (msg) {

case WM\_INITDIALOG: /\* меняем иконку диалога \*/

OldIcon=SetClassLong(hw,GCL\_HICON,(long)LoadIcon(hThisInstance,"Ex5\_Icon"));

return TRUE;

case WM\_COMMAND:

switch (LOWORD(wp)) {

case IDCANCEL: /\* посылается при закрытии диалога по [Esc]\*/

case IDC\_EXIT: /\* команда меню "Exit" \*/

DestroyWindow(hw);

break;

case IDC\_OPEN: /\* команда меню "Open" \*/

of.lStructSize=OPENFILENAME\_SIZE\_VERSION\_400A;

of.hwndOwner=hw;

of.lpstrFilter="All files (\*.\*)\0\*.\*\0";

of.lpstrCustomFilter=NULL; of.nMaxCustFilter=0;

of.nFilterIndex=1;

of.lpstrFile=filename; of.nMaxFile=MAX\_PATH;

of.lpstrFileTitle=NULL; of.nMaxFileTitle=0;

of.lpstrInitialDir=NULL;

of.Flags=OFN\_PATHMUSTEXIST|OFN\_FILEMUSTEXIST|OFN\_HIDEREADONLY;

if (!GetOpenFileName(&of)) break;

SetDlgItemText(hw,ID\_STATUS,filename);

/\* открываем файл \*/

hf=CreateFile(filename,GENERIC\_READ,0,NULL,OPEN\_EXISTING,0,NULL);

if (hf==INVALID\_HANDLE\_VALUE) {

MessageBox(hw,"Open failed","Error",MB\_ICONHAND|MB\_OK);

break;

}

len=GetFileSize(hf,NULL);

buf=(char\*)malloc(len+1); /\* доп. байт под символ-терминатор (0) \*/

if (!buf) {

MessageBox(hw,"Mem alloc failed","Error",MB\_ICONHAND|MB\_OK);

break;

}

ReadFile(hf,buf,len,&len1,NULL);

buf[len1]=0;

CloseHandle(hf);

SetDlgItemText(hw,ID\_EDIT,buf);

free(buf);

break;

case IDC\_SAVEAS: /\* команда меню "Save As" \*/

of.lStructSize=OPENFILENAME\_SIZE\_VERSION\_400A;

of.hwndOwner=hw;

of.lpstrFilter="All files (\*.\*)\0\*.\*\0";

of.lpstrCustomFilter=NULL; of.nMaxCustFilter=0;

of.nFilterIndex=1;

of.lpstrFile=filename; of.nMaxFile=MAX\_PATH;

of.lpstrFileTitle=NULL; of.nMaxFileTitle=0;

of.lpstrInitialDir=NULL;

of.Flags=OFN\_PATHMUSTEXIST|OFN\_OVERWRITEPROMPT|OFN\_HIDEREADONLY;

if (!GetSaveFileName(&of)) break;

case IDC\_SAVE: /\* команда меню "Save" \*/

if (lstrlen(filename)==0) {

/\* для нового файла - вызываем диалог "Save As" \*/

PostMessage(hw,WM\_COMMAND,IDC\_SAVEAS,lp);

break;

}

SetDlgItemText(hw,ID\_STATUS,filename);

/\* сохраняем файл \*/

hf=CreateFile(filename,GENERIC\_WRITE,0,NULL,OPEN\_ALWAYS,FILE\_ATTRIBUTE\_NORMAL,NULL);

if (hf==INVALID\_HANDLE\_VALUE) {

MessageBox(hw,"Save failed","Error",MB\_ICONHAND|MB\_OK);

break;

}

len=SendDlgItemMessage(hw,ID\_EDIT,WM\_GETTEXTLENGTH,0,0);

buf=(char\*)malloc(len+1); /\* доп. байт под символ-терминатор (0) \*/

GetDlgItemText(hw,ID\_EDIT,buf,len+1);

if (!buf) {

MessageBox(hw,"Mem alloc failed","Error",MB\_ICONHAND|MB\_OK);

break;

}

WriteFile(hf,buf,len,&len1,NULL);

CloseHandle(hf);

free(buf);

break;

case IDC\_ABOUT: /\* команда меню "About" \*/

MessageBox(hw,"Example N5 from http://dims.karelia.ru/win32","About",MB\_OK|MB\_ICONINFORMATION);

break;

}

return TRUE;

case WM\_DESTROY: /\* при закрытии окна восстанавливаем старую иконку \*/

SetClassLong(hw,GCL\_HICON,(long)OldIcon);

PostQuitMessage(0);

return TRUE;

}

return FALSE;

}

