Урок №3

Содержание

1. Диалоговое приложение

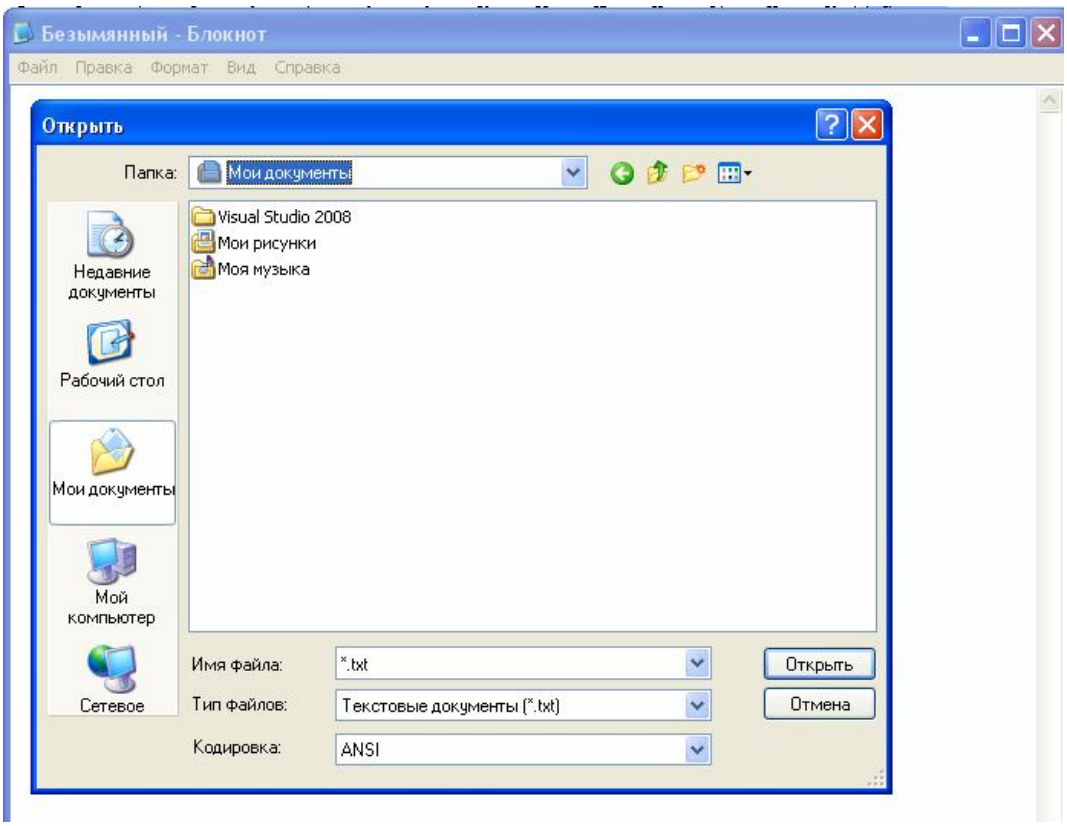
Диалог – это специальный тип окна, предоставляющий интерфейс для взаимодействия пользователя с приложением. Взаимодействие с пользователем осуществляется посредством элементов управления, которые являются дочерними окнами по отношению к диалоговому окну. **Элемент управления** представляет собой особый тип окна, предназначенный для ввода или вывода информации. Примером элемента

управления является кнопка, список, текстовое поле, переключатель и ряд других элементов.

Диалоги бывают двух типов: **модальные** (**modal**) и **немодальные (modeless)**. В большинстве случаев используются модальные диалоги.

1.1. Модальный диалог

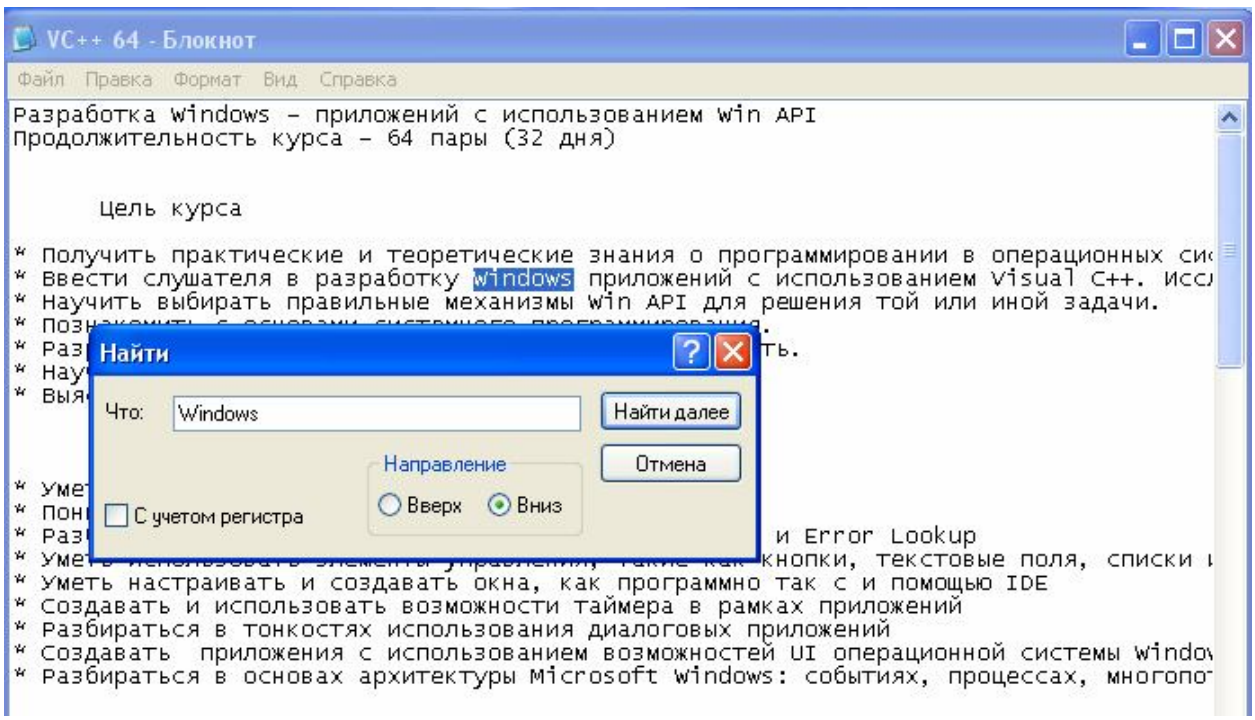
Особенность модального диалога состоит в том, что приложение, создав диалог, всегда дожидается его закрытия, после чего приложение продолжает свою работу. Модальный диалог не позволяет переключить ввод на другие окна, порожденные приложением. При этом пользователь может переключаться в другие программы, не закрыв диалоговое окно. Примером модального диалога может послужить диалоговое окно «Открыть» приложения «Блокнот».



Существует также специальный вид модальных диалоговых окон — **системные модальные (system modal)** окна, которые не позволяют переключаться даже в другие программы. Они сообщают о серьезных проблемах, и пользователь должен закрыть системное модальное окно, чтобы продолжить работу в Windows.

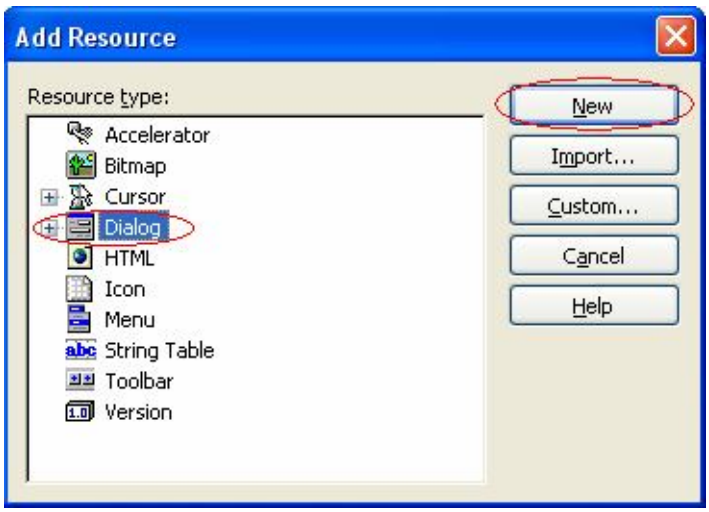
1.2. Немодальный диалог

Немодальный диалог не задерживает выполнение программы, то есть для ее продолжения не требуется завершение диалога. При этом разрешается переключение между диалогом и другими окнами приложения. Таким образом, немодальный диалог может получать и терять фокус ввода. Диалоги этого типа предпочтительней использовать в тех случаях, когда они содержат элементы управления, которые должны быть в любой момент доступны пользователю. Примером немодального диалога может послужить диалоговое окно «Найти» приложения «Блокнот».

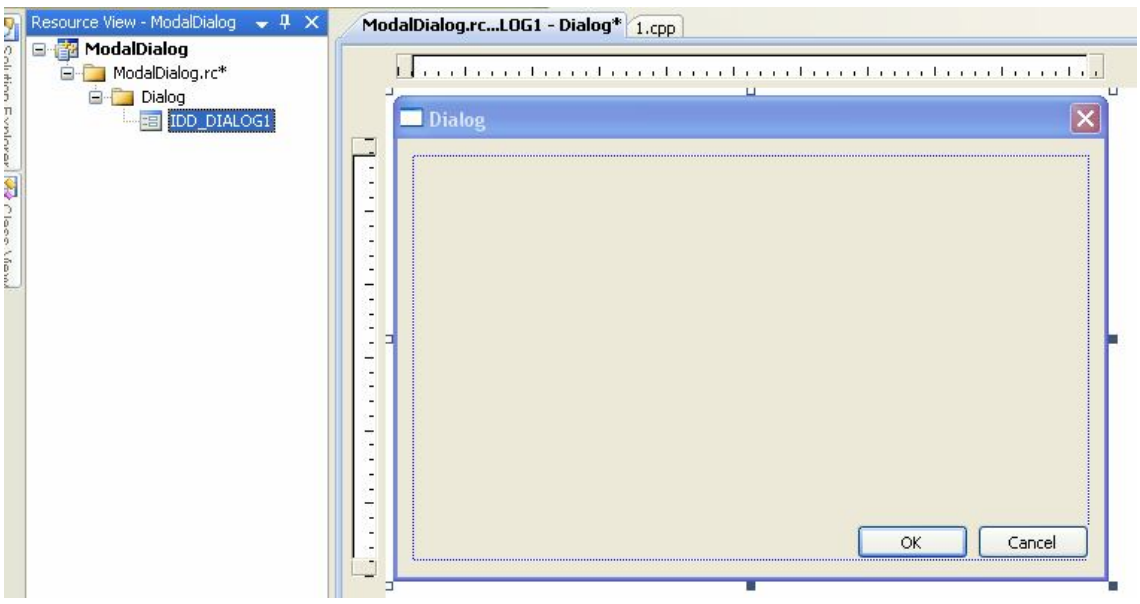


2. Создание диалогового приложения

Рассмотрим вопрос создания диалогового приложения, в котором диалоговое окно является главным окном приложения. Для этого, прежде всего, необходимо определить шаблон диалога в файле описания ресурсов. Для этого нужно активизировать вкладку **Resource View (<Ctrl><Shift><E>)**, в которой с помощью контекстного меню вызвать диалог добавления ресурса **Add -> Resource…**

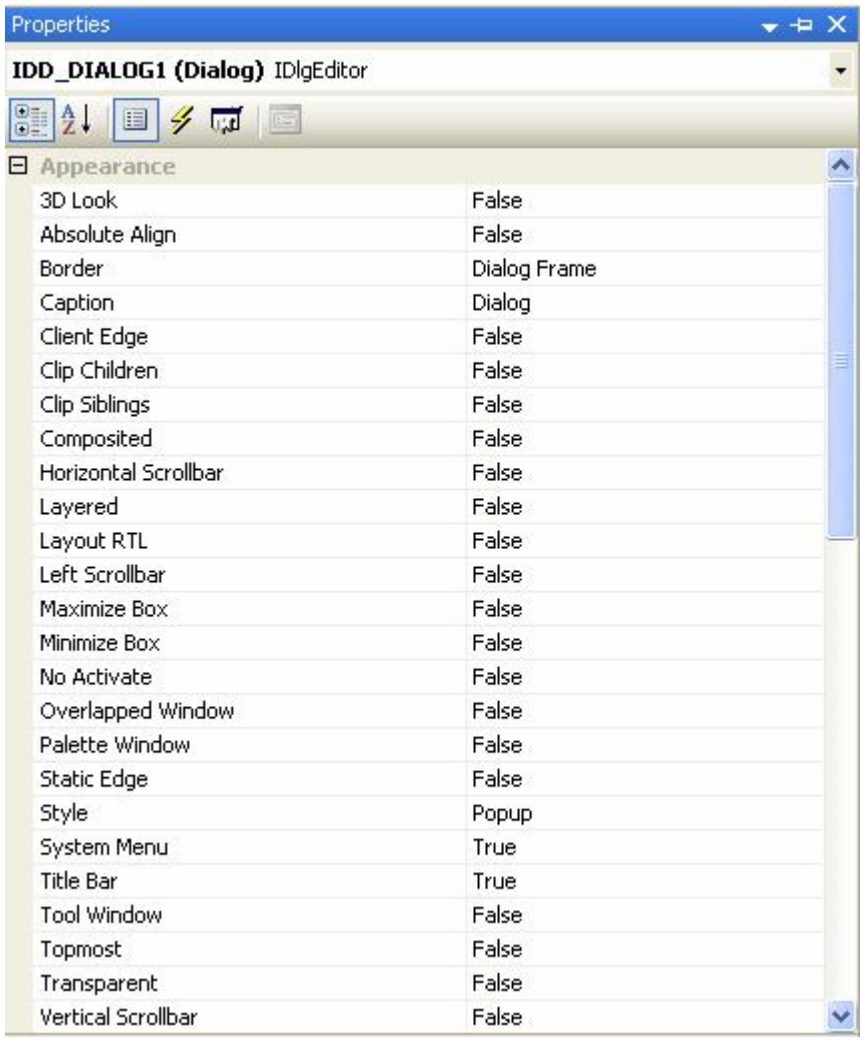


В этом диалоговом окне необходимо выбрать из списка **Dialog** и нажать кнопку **New**.



После добавления диалога в ресурсы приложения шаблон диалогового окна будет определён в файле описания ресурсов. При этом диалогу будет назначен идентификатор (например, **IDD_DIALOG1**), который впоследствии можно изменить на другой идентификатор, более точно отражающий семантику ресурса.

Рассмотрим некоторые свойства диалога. Для активизации окна свойств диалога следует выбрать **View -> Other Windows -> Properties Window (<Alt><Enter>).**



- Свойство **Border** со значением **Resizing** предоставляет возможность изменения размеров диалогового окна;
- Свойство **Caption** позволяет вывести название программы в заголовок окна;

- Свойство **Minimize Box** определяет наличие или отсутствие кнопки минимизации окна;
- Свойство **Maximize Box** определяет наличие или отсутствие кнопки полноэкранной развёртки окна;
- Свойством **System Menu** определяет наличие или отсутствие системного меню;
- Свойство **Title Bar** определяет наличие или отсутствие строки заголовка;
- Свойство **Center** со значением **True** позволяет расположить окно диалога в центре экрана при запуске программы.

2.1. Создание приложения на основе модального диалога

Приложение на основе модального диалога должно содержать как минимум две функции:

- **WinMain** — главную функцию, в которой создается модальное диалоговое окно программы;
- **DlgProc** — диалоговую процедуру, обеспечивающую обработку сообщений для диалогового окна программы.

```
#include <windows.h>
#include "resource.h"

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


int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInst,
                   LPSTR lpszCmdLine, int nCmdShow)
{
      // создаём главное окно приложения на основе модального диалога
      return DialogBox(hInstance, MAKEINTRESOURCE(IDD_DIALOG1), NULL,
                       DlgProc);
}

BOOL CALLBACK DlgProc(HWND hWnd, UINT message, WPARAM wp, LPARAM lp)
{
```

```
	switch(message)
	{
		case WM_CLOSE:
			EndDialog(hWnd, 0); // закрываем модальный диалог
			return TRUE;
	}
	return FALSE;
}
```

Исходный код приложения находится в папке **SOURCE/Modal Dialog**.

Необходимо отметить, что вышеприведенный пример не показывает механизм взаимодействия с модальным диалоговым окном (об этом речь пойдёт в последующих уроках), а лишь демонстрирует способ создания приложения на базе модального диалогового окна.

Как видно из вышеприведенного кода, создание модального диалога осуществляется функцией API **DialogBox**:

```
INT_PTR DialogBox(
    HINSTANCE hInstance, // дескриптор экземпляра приложения, содержащего
      // шаблон диалогового окна
    LPCTSTR lpTemplate, // указатель на строку, содержащую имя шаблона
      // диалогового окна
      HWND hWndParent, // дескриптор родительского окна
    DLGPROC lpDialogFunc // указатель на диалоговую процедуру
);
```

Особенность работы данной функции заключается в том, что она не возвращает управление до тех пор, пока функция обратного вызова (диалоговая процедура) не закроет модальное диалоговое окно, вызвав функцию API **EndDialog**:

```
BOOL EndDialog(
	HWND hDlg, // дескриптор диалогового окна
	INT_PTR nResult // значение, возвращаемое функцией DialogBox
);
```

Следует отметить, что диалоговая процедура во многом напоминает оконную процедуру. Она должна иметь спецификатор **CALLBACK**, по-

скольку вызывается операционной системой. Имя функции может быть произвольным. Диалоговая процедура принимает такой же набор параметров, что и обычная оконная процедура. Однако существуют некоторые различия между диалоговой процедурой и оконной процедурой.

- Оконная процедура возвращает значение типа **LRESULT**, а диалоговая процедура — значение типа **BOOL**.
- Если оконная процедура не обрабатывает какое-то сообщение, то она вызывает функцию **DefWindowProc**. Если диалоговая процедура не обрабатывает какое-то сообщение, то она возвращает значение **FALSE** (вызов стандартного обработчика сообщения).
- Если оконная процедура обрабатывает какое-то сообщение, то она возвращает значение **0**. Если диалоговая процедура обрабатывает какое-то сообщение, то она возвращает значение **TRUE** (запрет вызова стандартного обработчика сообщения).
- Для закрытия обычного приложения (не диалогового!!!) в оконной процедуре следует обработать сообщение **WM_DESTROY**. Для закрытия диалогового приложения необходимо в диалоговой процедуре обработать сообщение **WM_CLOSE**.

2.2. Создание приложения на основе немодального диалога

Немодальные диалоговые окна создаются с помощью функции API **CreateDialog**:

```
HWND CreateDialog(
      HINSTANCE hInstance, // дескриптор экземпляра приложения, содержащего
      // шаблон диалогового окна
      LPCTSTR lpTemplate, // указатель на строку, содержащую имя шаблона
      // диалогового окна
```

```
      HWND hWndParent, // дескриптор родительского окна
      DLGPROC lpDialogFunc // указатель на диалоговую процедуру
);
```

Функция **CreateDialog** в отличие от функции **DialogBox** сразу же возвращает дескриптор диалогового окна, не дожидаясь его закрытия.

Следует подчеркнуть, для того, чтобы немодальное окно появилось на экране необходимо в свойствах шаблона диалогового окна установить значение **True** для свойства **Visible**. Другим способом отображения немодального окна является вызов функции API **ShowWindow**.

Кроме того, в теле функции **WinMain** следует предусмотреть цикл обработки сообщений – ещё одно отличие от модального диалогового окна.

```
#include <windows.h>
#include "resource.h"

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

int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hPrev, LPSTR lpszCmdLine,
                        int nCmdShow)
{
      MSG msg;
      // создаём главное окно приложения на основе немодального диалога
      HWND hDialog = CreateDialog(hInst, MAKEINTRESOURCE(IDD_DIALOG1), NULL,
                                  DlgProc);
      // Отображаем окно
      ShowWindow(hDialog, nCmdShow);

      //Запускаем цикл обработки сообщений
      while(GetMessage(&msg, 0, 0, 0))
      {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
      }
      return msg.wParam;
}

BOOL CALLBACK DlgProc(HWND hWnd, UINT mes, WPARAM wp, LPARAM lp)
{

      switch(mes)
      {
            case WM_CLOSE:
                  // закрываем немодальный диалог
                  DestroyWindow(hWnd); // разрушаем окно
```

```
                // останавливаем цикл обработки сообщений
                PostQuitMessage(0);
                return TRUE;
        }
        return FALSE;
}
```

Исходный код приложения находится в папке **SOURCE/ModelessDialog**.

Необходимо отметить, что вышеприведенный пример не показывает механизм взаимодействия с немодальным диалоговым окном (об этом речь пойдёт в последующих уроках), а лишь демонстрирует способ создания приложения на базе немодального диалогового окна.

Как видно из вышеприведенного кода, для закрытия приложения, созданного на основе немодального диалога, необходимо в диалоговой процедуре при обработке сообщения **WM_CLOSE** предусмотреть вызов функции **API DestroyWindow** для разрушения диалогового окна, и кроме того, остановить цикл обработки сообщений с помощью функции **API PostQuitMessage.**

```
BOOL DestroyWindow(HWND hWnd);
```

3. Синхронные и асинхронные сообщения

Сообщения, посылаемые окну, могут быть **синхронными** и **асинхронными**.

Синхронные сообщения - это сообщения, которые Windows помещает в очередь сообщений приложения. В цикле очередное синхронное сообщение выбирается из очереди функцией **GetMessage**, затем передается Windows посредством функции **DispatchMessage**, после чего Windows отправляет синхронное сообщение оконной процедуре для последующей обработки.

В отличие от синхронных сообщений **асинхронные сообщения** передаются непосредственно оконной процедуре для немедленной обработки, минуя очередь сообщений.

К синхронным сообщениям относятся сообщения о событиях пользовательского ввода, например, клавиатурные сообщения и сообщения мыши. Кроме этого синхронными являются сообщения от таймера (**WM_TIMER**), сообщение о необходимости перерисовки клиентской области окна (**WM_PAINT**) и сообщение о выходе из программы (**WM_QUIT**). Остальные сообщения, как правило, являются асинхронными. Во многих случаях асинхронные сообщения являются результатом обработки синхронных сообщений. Например, когда **WinMain** вызывает функцию **CreateWindow**, Windows создает окно и для этого отправляет оконной процедуре асинхронное сообщение **WM_CREATE**. Когда **WinMain** вызывает **ShowWindow**, Windows отправляет оконной процедуре асинхронные сообщения **WM_SIZE** и **WM_SHOWWINDOW**. Когда **WinMain** вызывает **UpdateWindow**, Windows отправляет оконной процедуре асинхронное сообщение **WM_PAINT**.

4. Посылка сообщений

Существует множество задач, для решения которых необходимо уметь «вручную» посылать сообщения окнам, не дожидаясь их от операционной системы. Для этой цели используются функции API **SendMessage** и **PostMessage**.

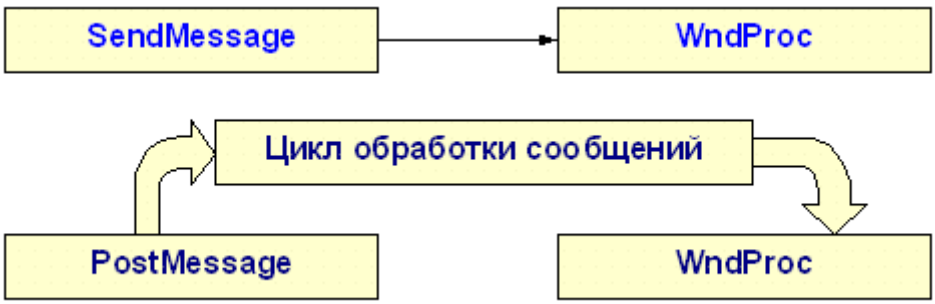
```
LRESULT SendMessage(
    HWND hWnd, // дескриптор окна, которому отправляется сообщение
    UINT Msg, // идентификатор сообщения
    WPARAM wParam, // дополнительная информация о сообщении
    LPARAM lParam  // дополнительная информация о сообщении
);
```

```
BOOL PostMessage(
    HWND hWnd, // дескриптор окна, которому отправляется сообщение
    UINT Msg, // идентификатор сообщения
    WPARAM wParam, // дополнительная информация о сообщении
    LPARAM lParam  // дополнительная информация о сообщении
);
```

Как видно, обе функции имеют одинаковую сигнатуру, а их сходство состоит в том, что они предназначены для посылки сообщения в окно некоторого приложения. При этом в 32-битных параметрах **wParam** и **lParam** передается дополнительная информация о сообщении. Это может быть целочисленное значение (**INT**, **LONG**, **DWORD** и т.д.), дескриптор ресурса (**HANDLE**, **HWND** и т.д.) либо адрес какого-либо объекта (как известно, адрес занимает 32-бита и вполне помещается в **wParam** или **lParam).** В последних двух случаях необходимо выполнять явное приведение типа к **WPARAM** или **LPARAM.**

Однако вышеприведенные функции имеют определённые отличия. Функция **SendMessage** вызывает оконную процедуру определенного окна и не завершает свою работу до тех пор, пока оконная процедура не обработает сообщение. Иначе говоря, выполнение программы не будет продолжено, пока сообщение не будет обработано. Оконная процедура, которой отправляется сообщение, может быть той же самой оконной процедурой, другой оконной процедурой той же программы или даже оконной процедурой другого приложения. Таким образом, функция **SendMessage** посылает асинхронное сообщение указанному окну.

Функция **PostMessage** посылает синхронное сообщение указанному окну. В отличие от **SendMessage**, функция **PostMessage** не вызывает явно оконную процедуру, а всего лишь помещает сообщение в очередь сообщений. Выполнение самой функции на этом завершается, и работа приложения может быть продолжена.



Следует отметить, если первый параметр вышерассмотренных функций имеет значение **HWND_BROADCAST**, то сообщение посылается всем окнам верхнего уровня, существующим в настоящий момент в системе.

Для сравнительного анализа работы функций **SendMessage** и **PostMessage** рассмотрим следующий код (исходный код приложения находится в папке **SOURCE/ Queued_and_nonqueued_messages**):

```
#include <windows.h>

LRESULT CALLBACK WindowProc(HWND, UINT, WPARAM, LPARAM);
TCHAR szClassWindow[] = TEXT("Каркасное приложение");

int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hPrevInst, LPSTR lpszCmdLine,
                        int nCmdShow)
{
      HWND hWnd;
      MSG lpMsg;
      WNDCLASSEX wcl;
      wcl.cbSize = sizeof(wcl);
      wcl.style = CS_HREDRAW | CS_VREDRAW;
      wcl.lpfnWndProc = WindowProc;
      wcl.cbClsExtra = 0;
      wcl.cbWndExtra = 0;
      wcl.hInstance = hInst;
      wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION);
      wcl.hCursor = LoadCursor(NULL, IDC_ARROW);
      wcl.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
      wcl.lpszMenuName = NULL;
      wcl.lpszClassName = szClassWindow;
      wcl.hIconSm = NULL;

      if (!RegisterClassEx(&wcl))
            return 0;

      hWnd = CreateWindowEx(0, szClassWindow,
            TEXT("Синхронные и асинхронные сообщения"),
            WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT, 500, 200,
            NULL, NULL, hInst, NULL);
```

```
	ShowWindow(hWnd, nCmdShow);
	UpdateWindow(hWnd);

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

	return lpMsg.wParam;
}

LRESULT CALLBACK WindowProc (HWND hWnd, UINT message, WPARAM wParam,
                             LPARAM lParam)
{

	switch(message)
	{
		case WM_DESTROY:
			PostQuitMessage(0);
			break;

		case WM_KEYDOWN:
			if(wParam == VK_UP)
			{
				HWND h = FindWindow(TEXT("SciCalc"),
					TEXT("Калькулятор"));
				if(h)
					SendMessage(h, WM_CLOSE, 0, 0);
			}
			else if(wParam == VK_DOWN)
			{
				HWND h = FindWindow(TEXT("SciCalc"),
					TEXT("Калькулятор"));
				if(h)
					PostMessage(h, WM_CLOSE, 0, 0);
			}
			else if(wParam == VK_LEFT)
			{
				HWND h = FindWindow(TEXT("SciCalc"),
					TEXT("Калькулятор"));
				if(h)
					SendMessage(h, WM_QUIT, 0, 0);
			}
			else if(wParam == VK_RIGHT)
			{
				HWND h = FindWindow(TEXT("SciCalc"),
					TEXT("Калькулятор"));
				if(h)
					PostMessage(h, WM_QUIT, 0, 0);
			}
			break;
		default:
			return DefWindowProc(hWnd, message, wParam, lParam);
	}
	return 0;
}
```

Анализируя вышеприведенный код, следует отметить, что при отправке сообщения **WM_CLOSE** главному окну приложения «Калькулятор» обе функции (**SendMessage** и **PostMessage**) дают одинаковый результат – приложение закрывается. Как известно, что при стандартной обработке сообщения **WM_CLOSE** вызывается функция **DestroyWindow** для разрушения главного окна приложения и его дочерних окон. Вызов этой функции, в свою очередь, приводит к посылке сообщения **WM_DESTROY**, при обработке которого оконная процедура обычно вызывает функцию **PostQuitMessage**. Как известно, вызов функции **PostQuitMessage** ставит сообщение **WM_QUIT** в очередь сообщений приложения. Впоследствии, функция **GetMessage** выбирает это сообщение из очереди, возвращает нулевое значение и завершает тем самым цикл обработки сообщений и, следовательно, приложение.

При отправке сообщения **WM_QUIT** главному окну приложения «Калькулятор» функции **PostMessage** и **SendMessage** дают различный результат. В первом случае (**PostMessage**) сообщение будет помещено в очередь сообщений приложения, что впоследствии приведёт к завершению приложения по вышеописанной причине. Во втором случае (**SendMessage**) сообщение будет направлено непосредственно оконной процедуре в обход очереди сообщений. В результате это не приведёт к закрытию приложения, так как сообщение **WM_QUIT** не ассоциируется с каким-либо окном и не предназначено для обработки оконной процедурой.

5. Общие сведения об элементах управления

Как было отмечено ранее, диалоговое приложение взаимодействует с пользователем посредством одного или нескольких элементов управления. Элементы управления выполняют основную функциональную нагрузку в диалоговом окне, которое является для них родитель-

ским. Windows поддерживает так называемые **базовые элементы управления**, включая кнопки (**Button**), флажки (**Check Box**), переключатели (**Radio Button**), списки (**List Box**), окна ввода (**Edit Control**), комбинированные списки (**Combo Box**), статические элементы (**Static Text**), полосы прокрутки (**Scroll Bar**), рамки (**Group Box**).

Помимо базовых элементов управления, которые поддерживались самыми ранними версиями Windows, в системе используется библиотека элементов управления общего пользования (**common control library**). Общие элементы управления, включенные в эту библиотеку, дополняют базовые элементы управления и позволяют придать приложениям более совершенный вид. К общим элементам управления относятся панель инструментов (**Toolbar**), окно подсказки (**Tooltip**), индикатор (**Progress Control**), счётчик (**Spin Control**), строка состояния (**Status Bar**) и другие.

Обычно элементы управления определяются в шаблоне диалогового окна на языке описания шаблона диалога. Одним из атрибутов описания элемента управления в шаблоне диалога является **идентификатор элемента управления.**

Каждый элемент управления, описанный в шаблоне диалога, реализуется Windows в виде окна соответствующего класса. Например, все кнопки относятся к классу **BUTTON**. Примерами других предопределённых классов для элементов управления являются **STATIC**, **LISTBOX**, **EDIT**, **COMBOBOX** и другие.

Как упоминалось ранее, элемент управления является дочерним окном по отношению к диалоговому окну. Как любое окно, элемент управления идентифицируется своим дескриптором типа **HWND**. Однако если элемент управления определен в шаблоне диалога, то приложению известен только его идентификатор. В то же время многие функции, работающие с элементом управления, принимают в качестве параметра его дескриптор. Для получения дескриптора элемента управления по его идентификатору используется функция API **GetDlgItem**:

```
HWND GetDlgItem(
       HWND hDlg, // дескриптор диалогового окна
       int nIDDlgItem  // идентификатор элемента управления
 );
```

В некоторых случаях возникает необходимость получения идентификатора элемента управления по его дескриптору. Для этого используется функция API **GetDlgCtrlID**:

```
int GetDlgCtrlID(
       HWND hwndCtrl // дескриптор элемента управления
   );
```

Чаще всего элементы управления определяются в шаблоне диалогового окна. Существует, однако, и альтернативный способ создания и размещения элемента управления при помощи функции **CreateWindowEx**, рассмотренной на одном из предыдущих занятий. В этом случае во втором параметре функции передается имя предопределенного оконного класса.

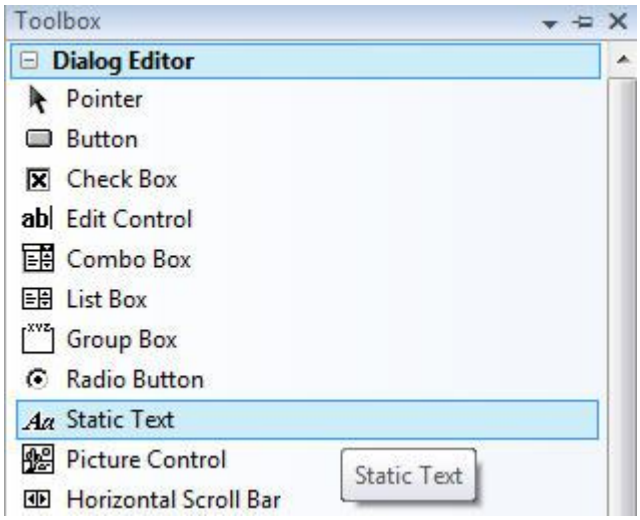
Элементы управления могут быть **разрешенными** (**enabled**) или **запрещенными** (**disabled**). По умолчанию все элементы управления имеют статус разрешенных элементов. Запрещенные элементы выводятся на экран серым цветом и не воспринимают пользовательский ввод с клавиатуры или от мыши. Изменение статуса элементов управления осуществляется при помощи функции API **EnableWindow**:

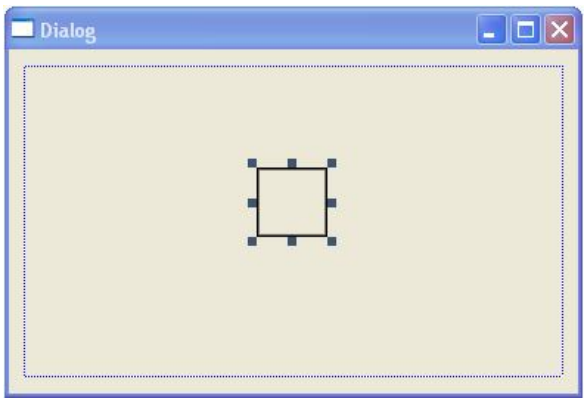
```
BOOL EnableWindow(
      HWND hWnd, // дескриптор окна
      BOOL bEnable // если данный параметр равен TRUE, то окно будет
      // разрешенным, в противном случае - запрещенным
);
```

5.1. Статический элемент управления Static Text

Статический элемент управления **StaticText** представляет собой средство описания чего-либо в диалоге и чаще всего просто отображается в виде текста.

Чтобы разместить на диалоге статический элемент управления следует активизировать окно **Toolbox** (<**Ctrl**><**Alt**><**X**>) и «перетащить» элемент управления на форму диалога.





Обычно всем статическим элементам управления назначается идентификатор **IDC_STATIC**:

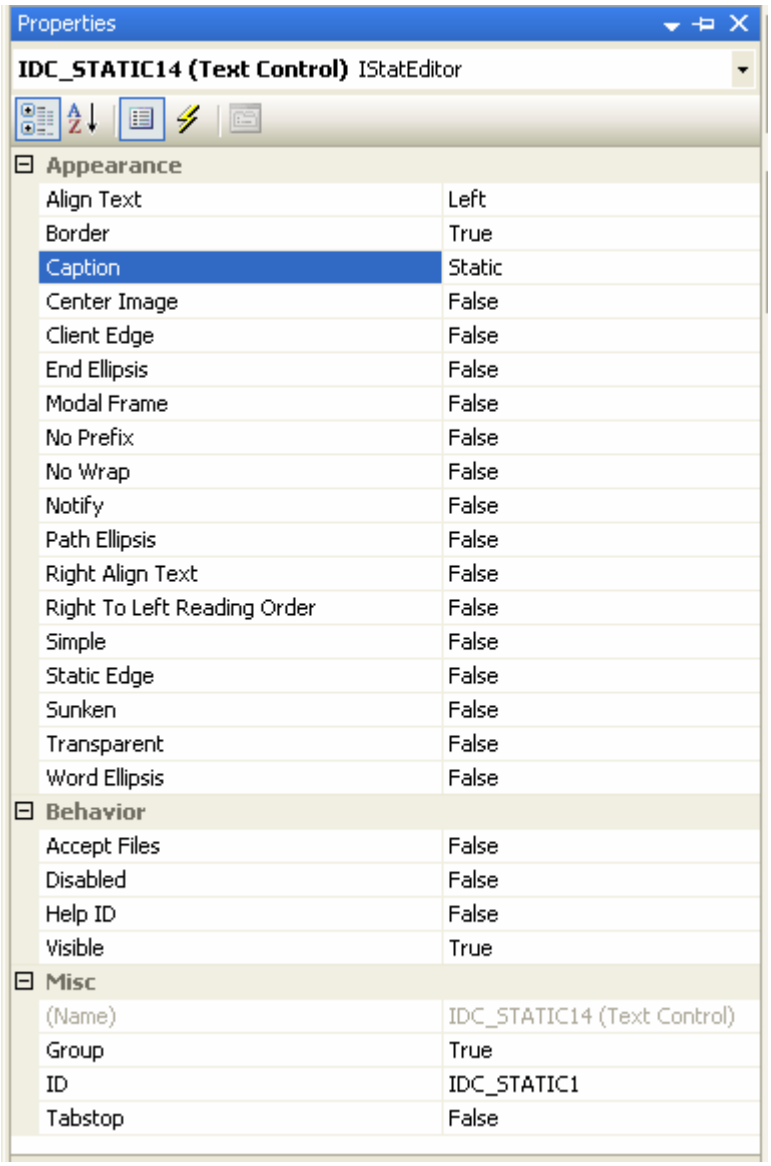
```
#define IDC_STATIC (-1)
```

Однако, если к «статику» необходимо будет обращаться в коде приложении, то ему следует назначить уникальный идентификатор, например **IDC_STATIC1**.

Рассмотрим некоторые свойства статического элемента управления **Static Text**.

- Свойство **Caption** содержит текстовую строку, которая будет отображаться внутри ограничивающего прямоугольника элемента **Static Text**. При этом в строке могут использоваться управляющие символы **\t** (табуляция) и **\n** (перевод строки).

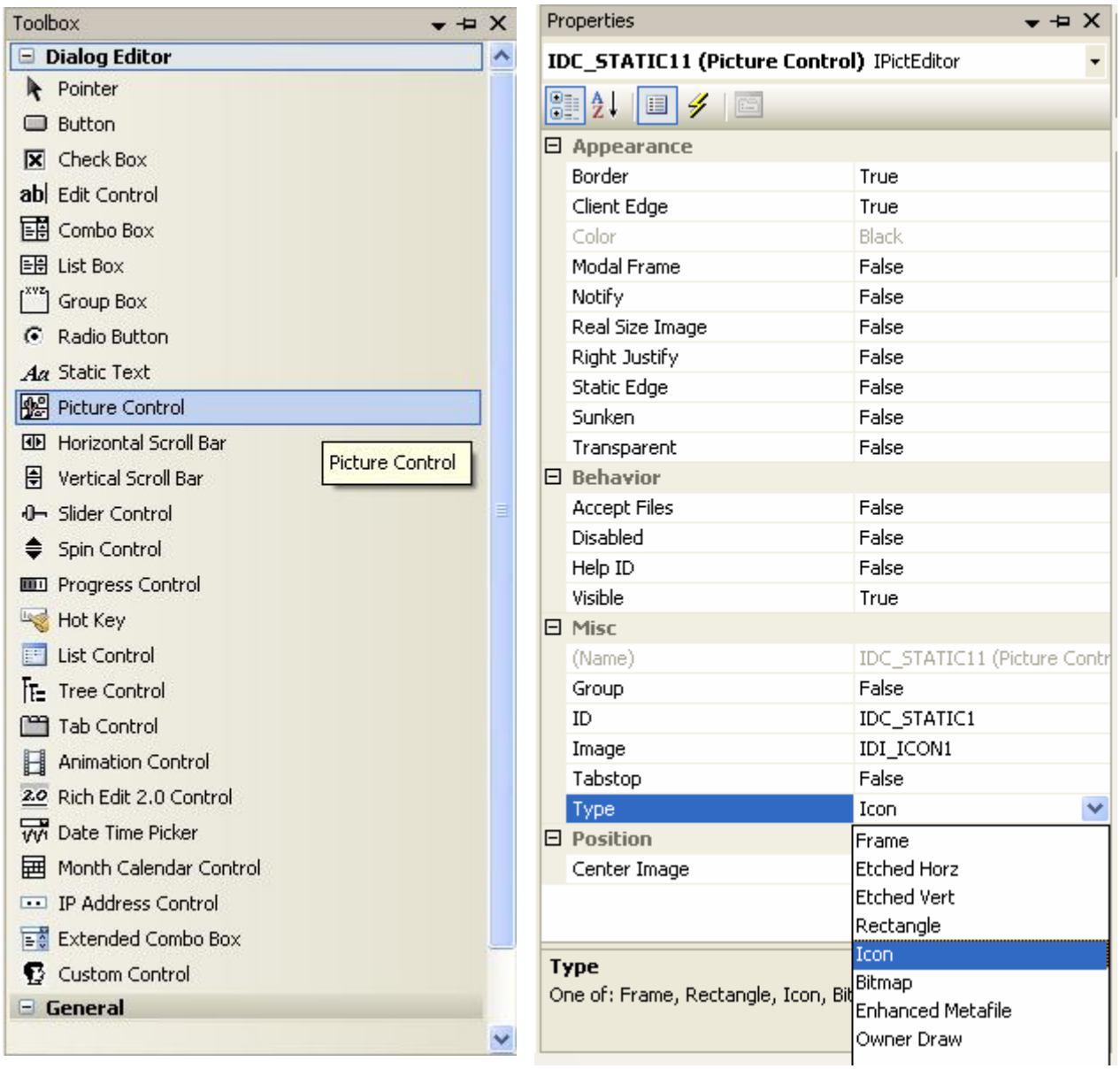
- Свойство **Border** со значением **True** позволяет установить тонкую рамку вокруг элемента управления.
- Свойство **Align text** позволяет задать выравнивание текста по горизонтали. Можно использовать значение **Left** (по умолчанию), **Center** или **Right**.
- Свойство **Center Image** позволяет задать центрирование текста по вертикали.



5.2. Статический элемент управления Picture Ctrl

Статический элемент управления **PictureCtrl** предназначен для размещения изображения на диалоговом окне.

Прежде чем поместить изображение (иконку или растровый образ) на форму диалога, необходимо сначала включить его в состав ресурсов проекта. После этого добавить на форму диалога элемент управления **PictureCtrl**.



В свойстве **Type** элемента управления следует задать тип изображения, например, **Icon**. В этом случае свойство **Image** станет доступным, что позволит выбрать из выпадающего списка нужный идентификатор изображения (например, **IDI_ICON1**).

Следующий пример демонстрирует использование статических элементов управления (исходный код приложения находится в папке **SOURCE/ UsingStatics)**:

```
#include <windows.h>
#include "resource.h"

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

HWND hStatic1, hStatic2;
TCHAR szCoordinates[20];
HINSTANCE hInst;
const int LEFT = 15, TOP = 110, WIDTH = 380, HEIGHT = 50;

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInst,
                   LPSTR lpszCmdLine, int nCmdShow)
{
      hInst = hInstance;
      // создаём главное окно приложения на основе модального диалога
      return DialogBox(hInstance, MAKEINTRESOURCE(IDD_DIALOG1), NULL,
                        DlgProc);
}

BOOL CALLBACK DlgProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{

      switch(message)
      {
            case WM_CLOSE:
                  EndDialog(hWnd, 0); // закрываем модальный диалог
                  return TRUE;
            // WM_INITDIALOG - данное сообщение приходит после создания
            // диалогового окна, но перед его отображением на экран
            case WM_INITDIALOG:
                  // получаем дескриптор статика, размещенного на диалоге
                  hStatic1 = GetDlgItem(hWnd, IDC_STATIC1);
                  //создаём статик с помощью CreateWindowEx
                  hStatic2 = CreateWindowEx(0, TEXT("STATIC"), 0,
                              WS_CHILD | WS_VISIBLE | WS_BORDER | SS_CENTER |
                              WS_EX_CLIENTEDGE, LEFT, TOP, WIDTH, HEIGHT,
                              hWnd, 0, hInst, 0);
                  return TRUE;
            case WM_MOUSEMOVE:
                  // текущие координаты курсора мыши
                  wsprintf(szCoordinates, TEXT("X=%d  Y=%d"),
                              LOWORD(lParam), HIWORD(lParam));
```

```
                    // строка выводится на статик
                    SetWindowText(hStatic1, szCoordinates);
                    return TRUE;

              case WM_LBUTTONDOWN:
                    SetWindowText(hStatic2, TEXT("Нажата левая кнопка мыши"));
                    return TRUE;
              case WM_RBUTTONDOWN:
                    SetWindowText(hStatic2, TEXT("Нажата правая кнопка мыши"));
                    return TRUE;
        }
        return FALSE;
}
```

Следует отметить, что первым в диалоговую процедуру приходит сообщение **WM_INITDIALOG**. К этому моменту диалоговое окно уже создано, но не отображено на экран. В обработчике этого сообщения удобно инициализировать элементы управления, а также выполнять другие инициализирующие действия.

В данном приложении был использован ещё один статический элемент управления – **Group Box**, который представляет собой рамку с заголовком. Этот элемент обычно используется для визуального группирования других элементов и может содержать заголовок (название) группы.

6. Элемент управления «кнопка»

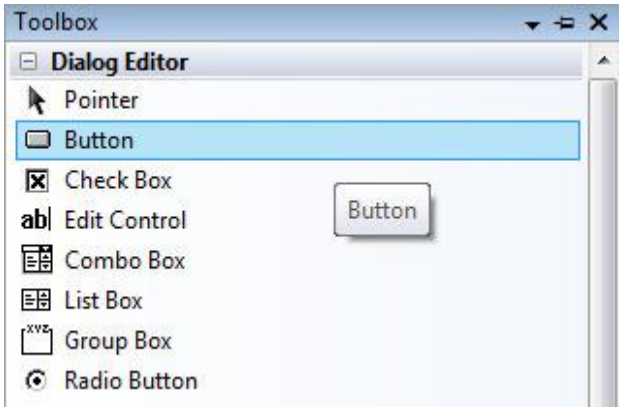
6.1. Обычная кнопка (Button)

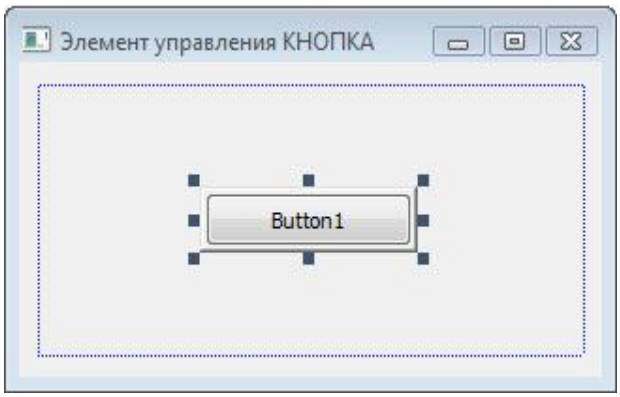
Создать кнопку на форме диалога можно двумя способами:

- с помощью средств интегрированной среды разработки **Microsoft Visual Studio**;
- посредством вызова функции **CreateWindowEx**.

При первом способе необходимо определить кнопку в шаблоне диалогового окна на языке описания шаблона диалога. Это произойдёт

автоматически, если активизировать окно **Toolbox** (<**Ctrl**><**Alt**><**X**>) и «перетащить» кнопку на форму диалога.

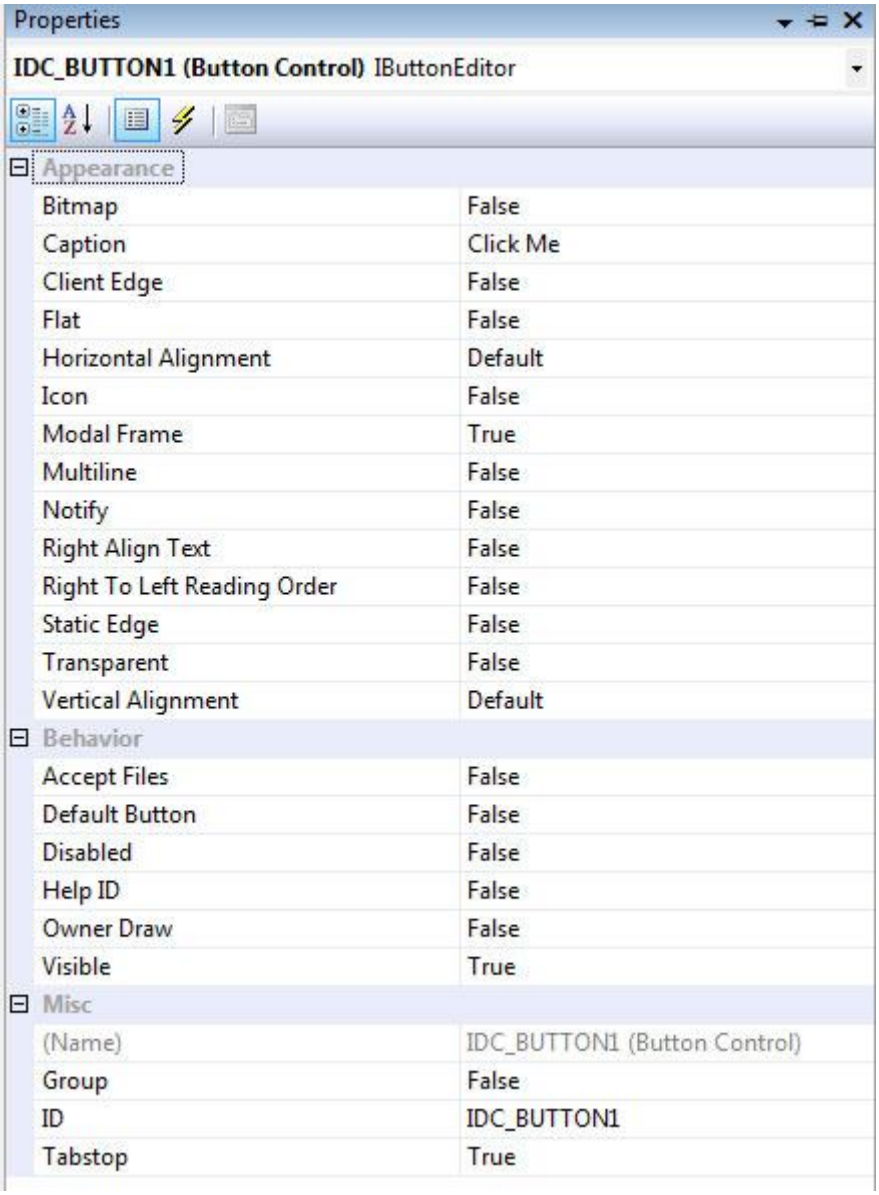




После размещения кнопки на форме диалога ей назначается идентификатор (например, **IDC_BUTTON1**), который впоследствии можно изменить на другой идентификатор, отражающий семантику ресурса.

Рассмотрим некоторые свойства элемента управления **Button**.

- Свойство **Caption** содержит текстовую строку, которая будет отображаться внутри ограничивающего прямоугольника элемента **Button**. При этом в строке могут использоваться управляющие символы **\t** (табуляция) и **\n** (перевод строки).
- Свойство **Multiline** позволяет располагать текст в несколько строк.
- Свойства **Horizontal Alignment** и **Vertical Alignment** позволяют выбрать вариант выравнивания текста внутри ограничивающего прямоугольника.
- Свойство **Flat** позволяет создать плоскую кнопку.
- Свойство **Default Button** назначает кнопке атрибут «применяемая по умолчанию».
- Свойства **Icon** или **Bitmap** позволяют указать, что вместо текста на кнопке будет отображаться пиктограмма или растровый образ.
- При истинном значении свойства **Notify** кнопка будет отправлять уведомление **BN_CLICKED** родительскому окну (диалогу).



В качестве примера, демонстрирующего использование кнопок, рассмотрим следующее приложение, в котором при нажатии на кнопку «Старт» начинается «Слайд-шоу» (периодическая смена изображений), а при нажатии на кнопку «Стоп» «Слайд-шоу» останавливается (исходный код приложения находится в папке **SOURCE/ Button_IDE**).

```
#include <windows.h>
#include "resource.h"

HWND hStart, hStop, hPicture;
HBITMAP hBmp[5];

BOOL CALLBACK DlgProc(HWND, UINT, WPARAM, LPARAM);
```

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInst,
                        LPSTR lpszCmdLine, int nCmdShow)
{

      return DialogBox(hInstance, MAKEINTRESOURCE(IDD_DIALOG1), NULL,
                        DlgProc);
}

BOOL CALLBACK DlgProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
      switch(message)
      {
            case WM_CLOSE:
                  EndDialog(hWnd, 0);
                  return TRUE;

            case WM_INITDIALOG:
                  hStart = GetDlgItem(hWnd, IDC_START);
                  hStop = GetDlgItem(hWnd, IDC_STOP);
                  hPicture = GetDlgItem(hWnd, IDC_PICTURE);
                  for(int i = 0; i < 5; i++)
                        hBmp[i] = LoadBitmap(GetModuleHandle(0),
                                    MAKEINTRESOURCE(IDB_BITMAP1 + i));
                  return TRUE;

            case WM_COMMAND:
                  if(LOWORD(wParam) == IDC_START)
                  {
                        SetTimer(hWnd, 1, 1000, 0);
                        EnableWindow(hStart, 0);
                        EnableWindow(hStop, 1);
                        SetFocus(hStop);
                  }
                  else if(LOWORD(wParam) == IDC_STOP)
                  {
                        KillTimer(hWnd, 1);
                        EnableWindow(hStart, 1);
                        EnableWindow(hStop, 0);
                        SetFocus(hStart);
                  }
                  return TRUE;

            case WM_TIMER:
                  static int index = 0;
                  index++;
                  if(index > 4)
                        index = 0;
                  SendMessage(hPicture, STM_SETIMAGE, WPARAM(IMAGE_BITMAP),
                              LPARAM(hBmp[index]));
                  return TRUE;
      }
      return FALSE;
}
```

Анализируя вышеприведенный код, следует отметить, что при воздействии на элемент управления диалога (в данном случае, при нажатии

на кнопку) в диалоговую процедуру **DlgProc** поступает сообщение **WM_COMMAND,** в котором **LOWORD(wParam)** содержит идентификатор элемента управления, **HIWORD(wParam)** содержит код уведомления (в данном случае, **BN_CLICKED)**, а **lParam** – дескриптор элемента управления.

Следует подчеркнуть, если кнопка имеет фокус ввода, то текст на кнопке обводится штриховой линией, а нажатие и отпускание клавиши пробела имеет тот же эффект, что и щелчок мышью по кнопке. Существует программный способ перевода фокуса ввода на элемент управления. Для этой цели служит функция API **SetFocus**:

```
HWND SetFocus(
HWND hWnd // дескриптор окна, приобретающего клавиатурный ввод
);
```

Для получения дескриптора окна (элемента управления), обладающего фокусом ввода используется функция API **GetFocus**:

```
HWND GetFocus(VOID);
```

В ресурсах вышеприведенного приложения имеются растровые битовые образы **Bitmaps**. Следует отметить, что добавление растрового битового образа (BMP-файла) в ресурсы приложения выполняется аналогично добавлению курсора или иконки, описанному в предыдущем уроке. Для использования в программе растрового битового образа, его предварительно следует загрузить функцией API **LoadBitmap**:

```
HBITMAP LoadBitmap(
  HINSTANCE hInstance,   // дескриптор приложения
  LPCTSTR lpBitmapName   // имя растрового битового образа
);
```

Продолжая анализировать вышеприведенный код, следует обратить внимание, что для установки изображения на «статик» ему посылается сообщение **STM_SETIMAGE**, с которым в **WPARAM** передаётся значение **IMAGE_BITMAP**, а в **LPARAM –** дескриптор растрового битового образа.

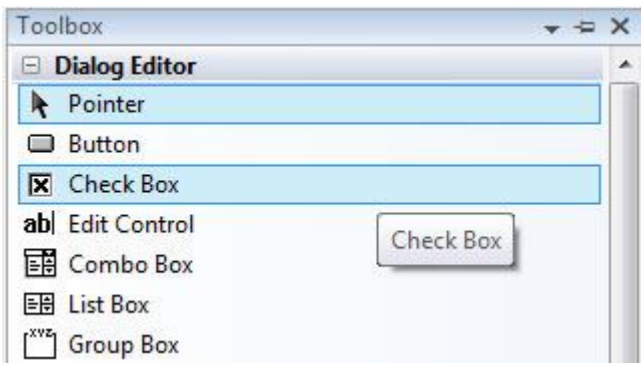
6.2. Флажок (Check Box)

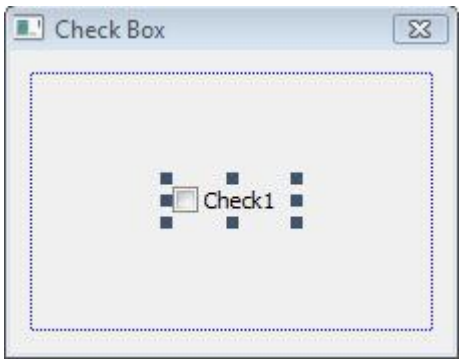
Данный элемент управления обычно применяется для установки или сброса определённых опций, независимых друг от друга. Флажок действует как двухпозиционный переключатель. Один щелчок вызывает появление контрольной отметки (галочки), а другой щелчок приводит к её исчезновению.

Создать **Check Box** на форме диалога можно двумя способами:

- с помощью средств интегрированной среды разработки **Microsoft Visual Studio**;
- посредством вызова функции **CreateWindowEx**.

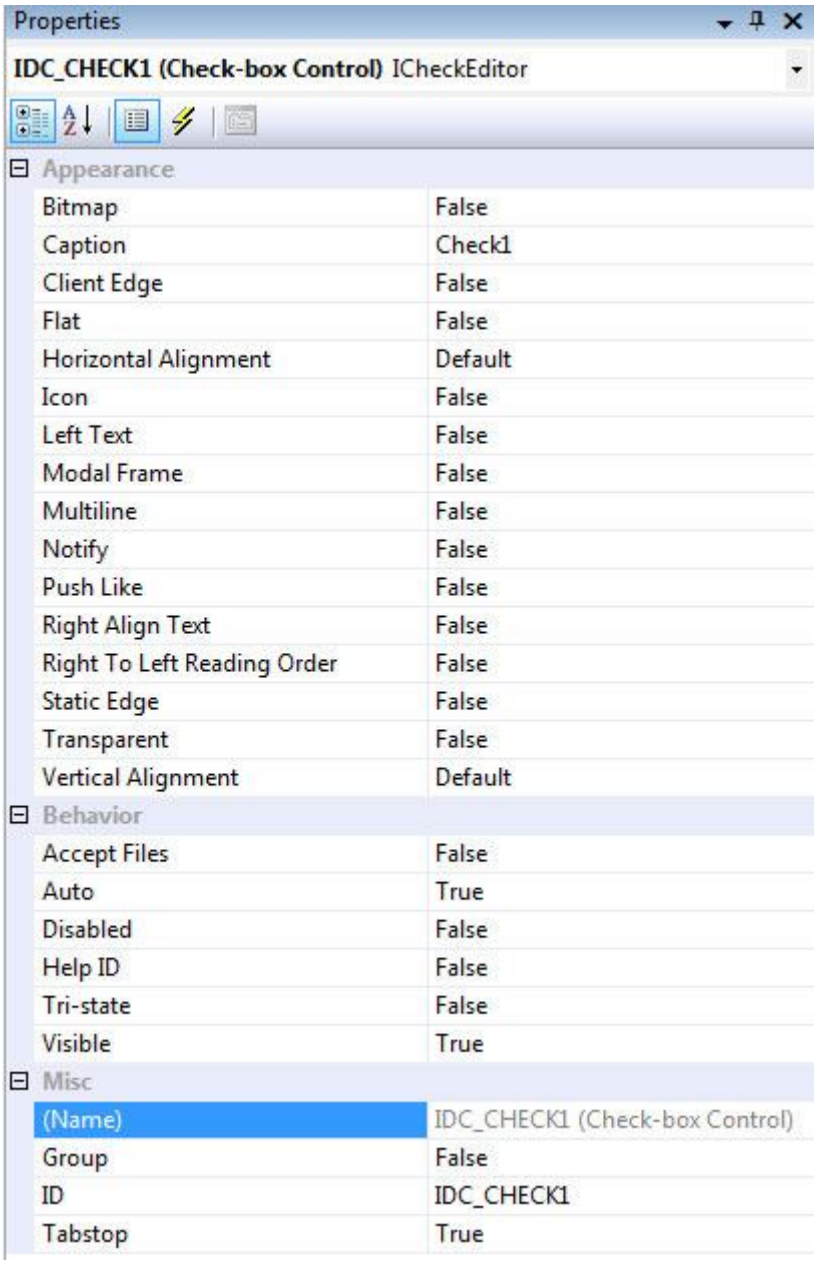
При первом способе необходимо определить **Check Box** в шаблоне диалогового окна на языке описания шаблона диалога. Это произойдёт автоматически, если активизировать окно **Toolbox** (<**Ctrl**><**Alt**><**X**>) и «перетащить» **Check Box** на форму диалога.





После размещения флажка на форме диалога ему назначается идентификатор (например, **IDC_CHECK1**), который впоследствии можно изменить на другой идентификатор, отражающий семантику ресурса.

Следует отметить, что элемент управления **Check Box** обладает тем же набором свойств, что и **Button**, а также располагает дополнительными свойствами, характерными только для него.



- Свойство **Auto** позволяет элементу управления отслеживать все щелчки мышью, и при этом элемент управления сам включает или

выключает контрольную отметку. Если же отключить свойство **Auto**, то управление флажком полностью возлагается на приложение.

- Свойство **Tri-state** используется для создания флажка, имеющего три состояния. Кроме состояний «установлен» и «сброшен» добавляется «неопределенное состояние», в котором флажок отображен в серой гамме. Серый цвет показывает пользователю, что выбор флажка не определен или не имеет отношения к текущей операции.
- Свойство **Push-like** изменяет внешний вид флажка так, что он выглядит как нажимаемая кнопка. Вместо установки галочки эта кнопка переходит в нажатое состояние и остается в нем до следующего щелчка мышью.

В качестве примера, демонстрирующего использование элемента управления **Check Box**, рассмотрим приложение, в котором на «статик» выводится текущее время, обновляемое через 1 секунду. При этом секунды будут отображаться в том случае, если установлен **CheckBox** (исходный код приложения находится в папке **SOURCE/ CheckBox**):

```
#include <windows.h>
#include <ctime>
#include <tchar.h>
#include "resource.h"

HWND hStart, hDateTime, hShowSeconds;

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInst,
                        LPSTR lpszCmdLine, int nCmdShow)
{
      return DialogBox(hInstance, MAKEINTRESOURCE(IDD_DIALOG1), NULL,
                        DlgProc);
}


BOOL CALLBACK DlgProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
      switch(message)
      {
```

```
            case WM_CLOSE:
                  EndDialog(hWnd, 0);
                  return TRUE;

            case WM_INITDIALOG:
                  hStart = GetDlgItem(hWnd, IDC_START);
                  hDateTime = GetDlgItem(hWnd, IDC_DATE_TIME);
                  hShowSeconds = GetDlgItem(hWnd, IDC_SHOW_SECONDS);
                  SendMessage(hShowSeconds, BM_SETCHECK,
                              WPARAM(BST_CHECKED), 0);
                  return TRUE;

            case WM_COMMAND:
                  if(LOWORD(wParam) == IDC_START)
                  {
                        TCHAR str[10];
                        GetWindowText(hStart, str, 10);
                        if(!lstrcmp(str, TEXT("Start")))
                        {
                              SetTimer(hWnd, 1, 1000, 0);
                              SetWindowText(hStart, TEXT("Stop"));
                        }
                        else if(!lstrcmp(str, TEXT("Stop")))
                        {
                              KillTimer(hWnd, 1);
                              SetWindowText(hStart, TEXT("Start"));
                              SetWindowText(hDateTime, 0);
                        }
                  }
                  return TRUE;

            case WM_TIMER:
                  {
                        static time_t t;
                        static TCHAR str[100];
                        t = time(NULL);
                        struct tm DateTime = *(localtime(&t));
                        LRESULT lResult = SendMessage(hShowSeconds,
                                          BM_GETCHECK, 0, 0);
                        if(lResult == BST_CHECKED)
                              _tcsftime(str, 100,
                              TEXT("%H:%M:%S  %d.%m.%Y  %A"), &DateTime);
                        else
                              _tcsftime(str, 100,
                              TEXT("%H:%M  %d.%m.%Y  %A"), &DateTime);
                        SetWindowText(hDateTime, str);
                  }
                  return TRUE;
      }
      return FALSE;
}
```

Анализируя вышеприведенный код, следует отметить, для того, чтобы перевести **Check Box** в некоторое состояние, ему необходимо от-

править сообщение **BM_SETCHECK**, передав в **WPARAM** одно из следующих значений:

- **BST_CHECKED** - установить отметку;
- **BST_UNCHECKED** – снять отметку;
- **BST_INDETERMINATE** - установить неопределенное состояние.

Существует альтернативный способ программной инициализации состояния элемента управления **Check Box**. Для этого используется функция API **CheckDlgButton**:

```
BOOL CheckDlgButton(
    HWND hDlg, // дескриптор диалога, содержащего кнопку (флажок)
    int nIDButton, // идентификатор элемента управления (флажка)
    UINT uCheck // состояние флажка - одно из вышеперечисленных значений
);
```

Для получения состояния флажка следует ему послать сообщение **BM_GETCHECK**. В этом случае **SendMessage** вернёт одно из вышеперечисленных значений.

Альтернативным способом получения состояния флажка является вызов функции API **IsDlgButtonChecked**:

```
UINT IsDlgButtonChecked(
     HWND hDlg, // дескриптор диалога, содержащего кнопку (флажок)
     int nIDButton // идентификатор элемента управления (флажка)
);
```

6.3. Переключатель (Radio Button)

Данный элемент управления обычно применяется для представления в окне множества взаимоисключающих опций, из которых можно

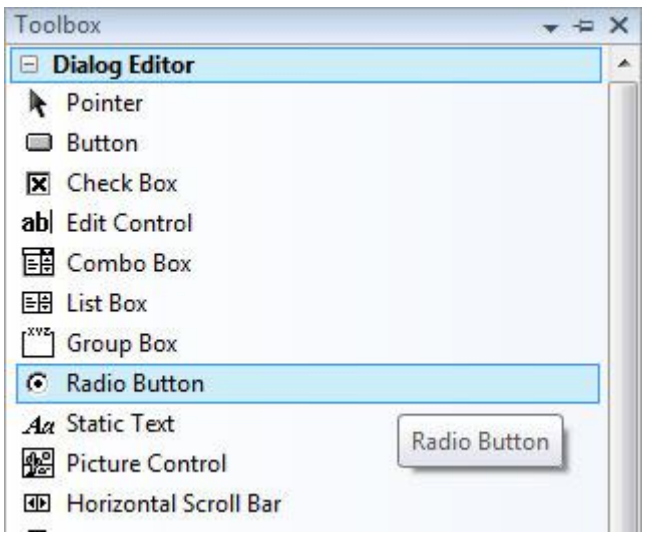
выбрать только одну. В отличие от флажков, повторный щелчок на переключателе не меняет его состояние.

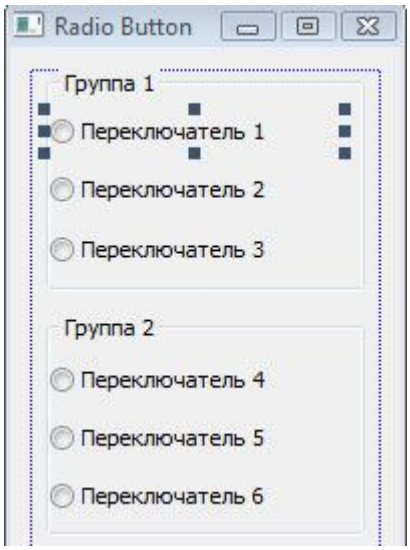
Создать **Radio Button** на форме диалога можно, как обычно, двумя способами:

- с помощью средств интегрированной среды разработки **Microsoft Visual Studio**;
- посредством вызова функции **CreateWindowEx**.

При первом способе необходимо определить **Radio Button** в шаблоне диалогового окна на языке описания шаблона диалога. Это произойдёт автоматически, если активизировать окно **Toolbox** (<**Ctrl**><**Alt**><**X**>) и «перетащить» **Radio Button** на форму диалога.

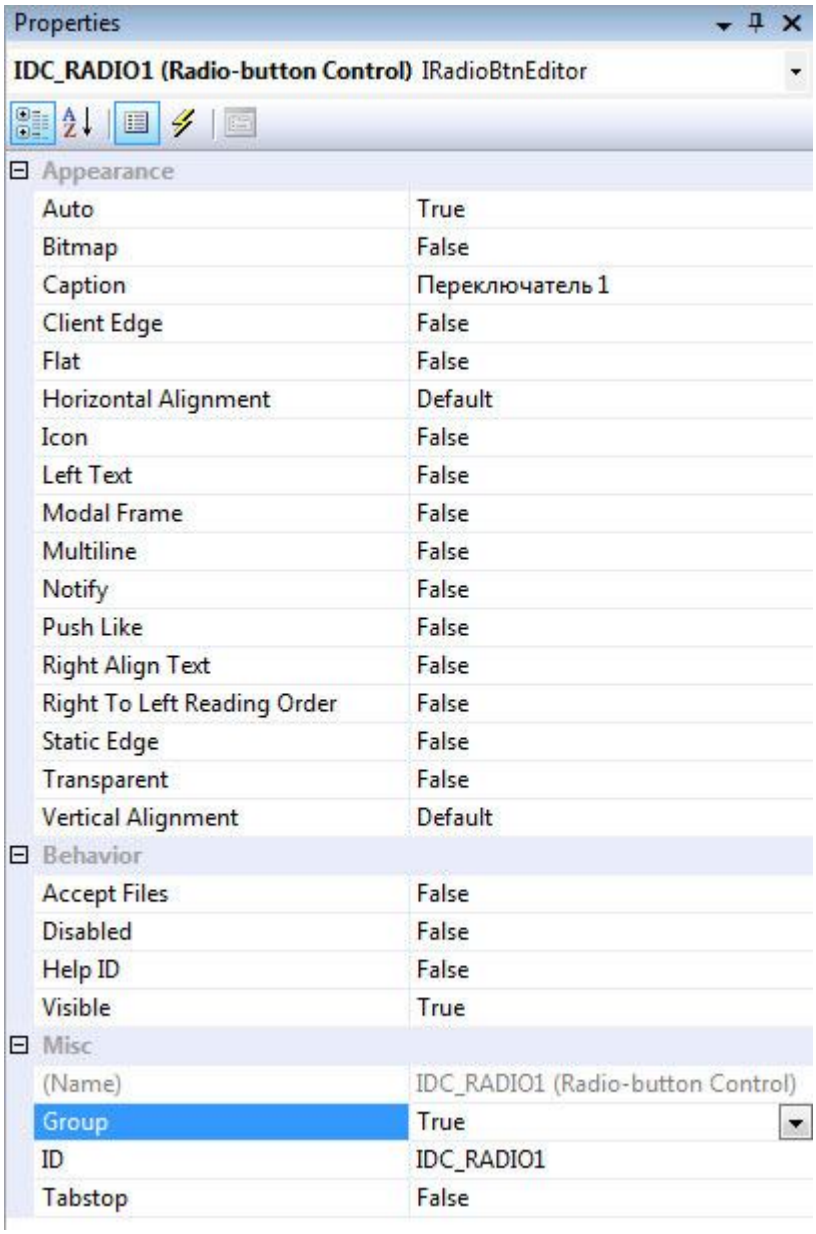
После размещения переключателя на форме диалога ему назначается идентификатор (например, **IDC_RADIO1**), который впоследствии можно изменить на идентификатор, отражающий семантику ресурса.





Следует отметить, что элемент управления **Radio Button** обладает тем же набором свойств, что и **Check Box**, но, кроме того, имеет важное свойство **Group**. Для первого переключателя в группе связанных взаимоисключающих переключателей нужно обязательно установить значение свойства **Group**, равным **True**. Все последующие переключатели (в файле описания ресурсов) со значением свойства **Group**, равным **False**,

считаются принадлежащими к этой группе. Если в последовательности описаний элементов управления встречается переключатель со значением свойства **Group**, равным **True**, считается, что он начинает новую группу элементов **Radio Button**.



В качестве примера, демонстрирующего использование элемента управления **Radio Button**, рассмотрим следующий код (исходный код приложения находится в папке **SOURCE/ RadioButton**):

```
#include <windows.h>
#include "resource.h"

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

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInst,
                   LPSTR lpszCmdLine, int nCmdShow)
{
      return DialogBox(hInstance, MAKEINTRESOURCE(IDD_DIALOG1), NULL,
                        DlgProc);
}

BOOL CALLBACK DlgProc(HWND hWnd, UINT message, WPARAM wp, LPARAM lp)
{
      switch(message)
      {
            case WM_CLOSE:
                  EndDialog(hWnd, 0);
                  return TRUE;

            case  WM_INITDIALOG:
                  SendDlgItemMessage(hWnd, IDC_RADIO2, BM_SETCHECK,
                                    WPARAM(BST_CHECKED), 0);
                  SendDlgItemMessage(hWnd, IDC_RADIO5, BM_SETCHECK,
                                    WPARAM(BST_CHECKED), 0);
                  return TRUE;

            case WM_COMMAND:
                  if(LOWORD(wp) >= IDC_RADIO1 && LOWORD(wp) <= IDC_RADIO6)
                  {
                        TCHAR str[20];
                        wsprintf(str, TEXT("Переключатель %d"),
                                    LOWORD(wp) - IDC_RADIO1 + 1);
                        SetWindowText(hWnd, str);
                  }
                  else if(LOWORD(wp)==IDC_BUTTON1)
                  {
                        TCHAR str[100] =
                              TEXT("Выбраны следующие переключатели:\n");

                        // Анализируем первую группу переключателей
                        LRESULT result = SendDlgItemMessage(hWnd, IDC_RADIO1,
                                          BM_GETCHECK, 0, 0);
                        if(result == BST_CHECKED)
                              lstrcat(str, TEXT("Радио №1\n"));
                        else
                        {
                              result = SendDlgItemMessage(hWnd, IDC_RADIO2,
                                          BM_GETCHECK, 0, 0);
                              if(result == BST_CHECKED)
                                    lstrcat(str, TEXT("Радио №2\n"));
                              else
                              {
                                    result = SendDlgItemMessage(hWnd,
                                          IDC_RADIO3, BM_GETCHECK, 0, 0);
                                    if(result == BST_CHECKED)
                                          lstrcat(str, TEXT("Радио №3\n"));
                              }
                        }
```

```
                        // Анализируем вторую группу переключателей
                        result = SendDlgItemMessage(hWnd, IDC_RADIO4,
                                                    BM_GETCHECK, 0, 0);
                        if(result == BST_CHECKED)
                              lstrcat(str, TEXT("Радио №4\n"));
                        else
                        {
                              result = SendDlgItemMessage(hWnd, IDC_RADIO5,
                                                    BM_GETCHECK, 0, 0);
                              if(result == BST_CHECKED)
                                    lstrcat(str, TEXT("Радио №5\n"));
                              else
                              {
                                    result = SendDlgItemMessage(hWnd,
                                          IDC_RADIO6, BM_GETCHECK, 0, 0);
                                    if(result == BST_CHECKED)
                                          lstrcat(str, TEXT("Радио №6\n"));
                              }
                        }

                        MessageBox(hWnd, str, TEXT("Radio Button"),
                                    MB_OK | MB_ICONINFORMATION);
                  }
                  return TRUE;
      }
      return FALSE;
}
```

Анализируя вышеприведенный код, следует отметить, для того, чтобы перевести **Radio Button** в некоторое состояние, ему необходимо отправить сообщение **BM_SETCHECK**, передав в **WPARAM** одно из следующих значений:

- **BST_CHECKED** - установить отметку;
- **BST_UNCHECKED** – снять отметку;
- **BST_INDETERMINATE** - установить неопределенное состояние.

Существует альтернативный способ выбора переключателя. Для этого используется функция API **CheckRadioButton**:

```
BOOL CheckRadioButton(
    HWND hDlg, // дескриптор диалога, содержащего кнопку (переключатель)
    int nIDFirstButton, // идентификатор первого переключателя в группе
    int nIDLastButton, // идентификатор последнего переключателя в группе
    int nIDCheckButton // идентификатор выбираемого переключателя
);
```

Данная функция помечает указанный переключатель в группе, удаляя отметку со всех других переключателей этой же группы. Другими словами, функция **CheckRadioButton** посылает сообщение **BM_SETCHECK** каждому переключателю указанной группы. При этом выбираемому переключателю в **WPARAM** передается **BST_CHECKED,** а остальным **- BST_UNCHECKED.**

Для получения состояния переключателя следует ему послать сообщение **BM_GETCHECK**. В этом случае **SendMessage** (либо **SendDlgItemMessage**) вернёт одно из вышеперечисленных значений.

Альтернативным способом получения состояния переключателя является вызов функции API **IsDlgButtonChecked**, рассмотренной ранее.

Кроме того, в вышеприведенном коде неоднократно использовалась функция API **SendDlgItemMessage**, предназначенная для отправки сообщения элементам управления.

```
LRESULT SendDlgItemMessage(
    HWND hDlg, // дескриптор диалога, содержащего элемент управления
    int nIDDlgItem, // идентификатор дочернего окна (элемента управления),
                    // которому отправляется сообщение
    UINT Msg, // идентификатор сообщения
    WPARAM wParam, // дополнительная информация о сообщении
    LPARAM lParam  // дополнительная информация о сообщении
);
```

Домашнее задание

1. Разработать приложение, созданное на основе диалогового окна, и обладающее следующей функциональностью.

- Пользователь «щелкает» левой кнопкой мыши по форме диалога и, не отпуская кнопку, ведёт по ней мышку, а в момент отпускания кнопки по полученным координатам прямоугольника (как известно, двух точек на плоскости достаточно для создания прямоугольника) создаётся «статик», который содержит свой порядковый номер (имеется в виду порядок появления «статика» на форме).
- Минимальный размер «статика» составляет 10x10, а при попытке создания элемента управления меньших размеров пользователь должен увидеть соответствующее предупреждение.
- При щелчке правой кнопкой мыши над поверхностью «статика» в заголовке окна должна появиться информация о статике (порядковый номер «статика», ширина и высота, а также координаты «статика» относительно родительского окна). В случае если в точке щелчка находится несколько «статиков», то предпочтение отдается «статику» с наибольшим порядковым номером.
- При двойном щелчке левой кнопки мыши над поверхностью «статика» он должен исчезнуть с формы (для этого можно воспользоваться функцией **DestroyWindow**, вызывая её для соответствующего объекта «статика»). В случае если в точке щелчка находится несколько «статиков», то предпочтение отдается «статику» с наименьшим порядковым номером.

При разработке приложения рекомендуется использовать библиотеку STL.

2. Написать игру «Крестики-нолики», учитывая следующие требования:

- игровое поле размером 3х3 должно состоять из кнопок;
- при нажатии на кнопку, на ней должна отобразиться картинка (крестик или нолик). Для установки изображения на кнопку ей нужно отправить сообщение **BM_SETIMAGE**, с которым в **WPARAM** передать значение **IMAGE_BITMAP**, а в **LPARAM –** дескриптор растрового битового образа (и главное не забыть про свойство **Bitmap!!!**);
- необходимо предотвращать попытку поставить крестик или нолик на занятую клетку;
- предоставить пользователю право выбора первого хода, используя флажок;
- предусмотреть возможность выбора уровня сложности, используя переключатели;
- предусмотреть кнопку «Начать новую игру».

3. Разработать приложение «убегающий статик». Суть приложения: на форме диалогового окна расположен статический элемент управления. Пользователь пытается подвести курсор мыши к «статику», и, если курсор находится близко со «статиком», элемент управления «убегает». Предусмотреть перемещение «статика» только в пределах диалогового окна.