Федеральное агентство по образованию
Озерский технологический институт (филиал)
ГОУ ВПО «Московский инженерно-физический институт
(государственный университет)»

Вл. Пономарёв

Microsoft Visual C++ 6.0
Приложение Windows с нуля

Учебное пособие

Озерск, 2006

УДК 681.3.06
П 56
Пономарёв Вл.
Microsoft Visual C++ 6.0. Приложение Windows с нуля. Учебное пособие.
Редакция 1. 28.03.2006.
Озерск: ОТИ МИФИ, 2006. — 59 с., ил.

Учебное пособие предназначено для изучения основ программирования приложений Windows при помощи Microsoft Visual C++ 6.0 студентами-программистами, обучающимися по специальности 230105 — «Программное обеспечение вычислительной техники и автоматизированных систем».

ОТИ МИФИ, 2006

Содержание

Введение **4**

Приложения Windows **5**

Окна Windows **6**

Ресурсы приложения Windows **7**

Приложение диалогового типа **8**

- Проект 8
- Ресурсы 9
- Глобальные объекты 11
- Оконная процедура 12
- Главное окно 13
- Стили окна 17
- Интерфейс 18
- Дочерние окна 20
- События мыши 24
- Шрифты 27
- OWNDC 28
- Надписи 29
- Кнопки 34
- Функциональность программы 35
  - Класс файла инициализации 35
  - Класс игрового поля 39
  - Класс пути нового слова 40
  - Класс слов игры 42
  - Класс словаря 43
- Диалоги 51
  - Передача и возврат параметров диалога 54
  - Диалог «О программе...» 54
- Размеры и положение окна 55
- Акселераторы 55
- Заключение 56

Приложения **57**

- Стили окна Windows 57
- Классы элементов управления 58
- Константы сообщений 59

Введение

Программирование приложений значительно отличается от программирования алгоритмических задач. Прежде всего, приложение Windows обладает *интерфейсом*. Интерфейс складывается из *окон приложения*, на которых размещаются различные визуальные компоненты, называемые *элементами управления*. При помощи элементов управления приложение *взаимодействует* с пользователем, реагируя (или не реагируя) на его действия, которые он осуществляет при помощи мыши и клавиатуры.

Вторая важная особенность приложения Windows — в нем отсутствует прямая последовательность действий, которую можно описать в виде линейного алгоритма. Вместо этого программа состоит из отрывков, описывающих реакцию на действия пользователя. Поскольку последовательность действий пользователя может быть произвольной, программа должна учитывать это и правильно реагировать на любые возможные действия. Последовательно описывается только момент старта (инициализации) программы. Как только на экране появится главное окно программы, управление неявно переходит к операционной системе, и далее будет выполняться тот отрывок кода, который соответствует очередному действию пользователя.

Третья особенность программы для Windows — сообщения. Все, что происходит в системе и программе, описывается сообщениями, которые посылаются тому или иному окну программы. В обязанность программиста входит отслеживание сообщений и описание реакции на них.

Наконец, важнейшая особенность системы Windows — «всё есть окно». Окно приложения, элемент управления (такой, как кнопка, текст, список и т.д.), диалоговое окно — всё это *окна*, имеющие разные характеристики (свойства). Программист описывает интерфейс, создавая необходимые окна с требуемыми характеристиками. С некоторыми окнами связываются *специальные процедуры*, которые предназначены для обработки приходящих в них сообщений. В этих процедурах программист описывает реакцию на действия пользователя, а также на сообщения, пришедшие от системы, или посланные им самим.

Приложение Windows может быть создано при помощи среды Microsoft Visual C++ 6.0 различными способами.

1. Код пишется вручную, без использования мастеров.
2. Используется библиотека фундаментальных классов MFC (Microsoft Foundation Classes). Приложения, созданные при помощи MFC, наиболее надежные, но и требующие наибольших ресурсов. Библиотека описывает все необходимые программисту окна и обработку событий.
3. Используется библиотека шаблонов ATL (ActiveX Template Library). Основное назначение шаблонов — программирование ActiveX компонентов, в основе которых лежит технология COM-объектов.

В настоящем пособии описывается программирование приложения вручную. Основы использования библиотеки MFC описываются в документе «Приложение MFC с нуля». Использование библиотеки ATL рассматривается в курсе «Современные технологии программирования». Дополнительно следует использовать описание графических возможностей системы Windows. Основным источником информации для программирования в среде Visual C++ является библиотека MSDN (Microsoft Developer Network).

Приложения Windows

Приложение Windows всегда располагается внутри ограниченного пространства экрана (дисплея), называемого *окном*. Как правило, окно имеет прямоугольную форму, но это не обязательно.

Приложение Windows строится по одной из трех схем.

1. *Приложение диалогового типа*. Самое простое приложение, которое состоит из главного окна, внутри которого располагаются элементы управления. Такое приложение, как правило, не имеет меню, и поэтому окно содержит кнопку, при помощи которой приложение можно закрыть (но может и не содержать ее, так как окно может быть закрыто при помощи системного меню). Особенность главного окна приложения диалогового типа — нельзя изменить размер окна. Если бы это было разрешено, при изменении размера окна можно было бы скрыть часть элементов управления, которые располагаются в окне в определенных местах, а изменять это расположение не представляется разумным. Примером приложения диалогового типа может служить системный калькулятор.
2. *Приложение SDI*. SDI расшифровывается как Single-Document Interface (одно-документный интерфейс). Это означает, что приложению сопоставлен документ — данные, которые приложением могут быть обработаны, в том числе сохранены *в* или прочитаны *из* файла документа. Приложение SDI имеет меню, в котором перечисляются все функциональные возможности. Кроме того, обычно оно имеет также панель инструментов, которая содержит кнопки для быстрого доступа к основным функциям, а также панель (строку) состояния, в которой отображаются те или иные элементы текущего состояния программы и/или системы. Главное окно приложения может быть изменено в размерах. Главную часть окна занимает документ приложения, его графическое представление. Примерами приложений SDI являются простой редактор текста «Блокнот» и простой графический редактор «Paint».
3. *Приложение MDI*. MDI расшифровывается как Multiple-Document Interface (многодокументный интерфейс). Как следует из названия, приложению также сопоставлен документ, но в отличие от приложения SDI, приложение MDI может одновременно работать с несколькими документами. Такое приложение значительно отличается от двух предыдущих прежде всего тем, что главное окно приложения — это окно специального класса, которое не имеет пространства для размещения элементов управления. Вместо этого вся внутренность окна представляет собой область для размещения других окон — окон документов. Иначе говоря, окна документов находятся внутри главного окна приложения, и не могут быть вынесены из него. Окна документов имеют отличительную характеристику — признак дочернего окна MDI, из-за которой они всегда остаются внутри главного окна. Примерами приложений MDI являются такие приложения пакета Microsoft Office, как Microsoft Word, Microsoft Excel и другие. Приложения типа MDI являются наиболее сложными и функциональными.

Перед тем, как разрабатывать приложение Windows, нужно определиться с тем, каков будет его тип. При этом в первую очередь учитываются данные, для работы с которыми приложение создается (если таковые есть).

Окна Windows

Окно — главный элемент системы Windows (*Windows* — окна). Картинка, кнопка, флажок или список, не говоря уже об окнах сообщений, диалоговых окнах и основных окнах приложения — все это окна, различающиеся своими характеристиками и назначением.

Окно состоит из системной и клиентской частей. *Системная часть* — это заголовок, меню, граница. В заголовке располагается *системное*, или оконное меню, которое управляет основными действиями с окном — свернуть, развернуть, переместить, закрыть. При помощи границы можно изменить размер окна.

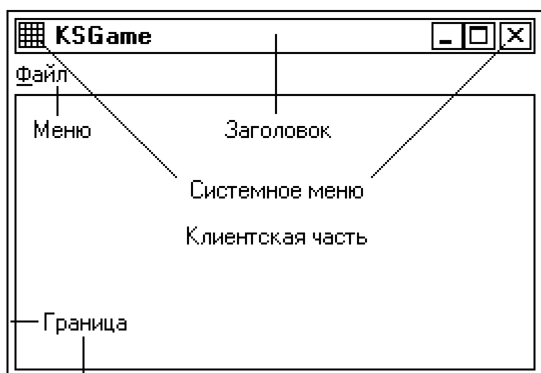

Рисунок 1. Главное окно приложения

Клиентская часть окна — это его внутренность, доступная для пользователя (программиста). В этой части располагается либо документ приложения, либо элементы управления или другие элементы, которые формируют внешний вид приложения.

Системной частью окна управляет операционная система. Клиентской частью окна управляет программист. Система посылает окну сообщения двух типов — системные и клиентские. Системные сообщения имеют в своем названии префикс *NC* (*non-client*), например, *WM_NCCREATE*. Аналогичное клиентское сообщение называется *WM_CREATE*.

Все окна можно разделить на следующие группы:

- *Перекрывающиеся* (*overlapped*) — главные окна приложений. Они имеют все атрибуты — заголовок, системное меню, границу, меню, панели инструментов. Описываются стилем *WS_OVERLAPPEDWINDOW*.
- *Всплывающие* (*pop-up*) — вспомогательные окна, используемые для диалогов, сообщений и других временных целей. У них может отсутствовать заголовок и, как правило, нет меню. Описываются стилем *WS_POPUP*.
- *Дочерние окна* (*child*) — содержащиеся внутри главных или всплывающих окон приложения. Не имеют заголовка. Их назначение — поделить клиентскую часть родительского окна на функциональные области. К ним же относятся и элементы управления. Описываются стилем *WS_CHILD*.
- *Слоёные* (*layered*) — окна, имеющие произвольную форму и позволяющие получить различные визуальные эффекты, например, прозрачность. Описываются расширенным стилем *WS_EX_LAYERED*.
- *Окна сообщений* (*message-only*) — предназначены исключительно для передачи сообщений. Описываются константой *HWND_MESSAGE*, назначаемой параметру *hWndParent* функции создания окна.

Ресурсы приложения Windows

Приложению Windows могут быть сопоставлены ресурсы. *Ресурс* — это некоторые двоичные данные. Различают стандартные и пользовательские (*defined*, *custom*) ресурсы. Стандартные ресурсы — это:

- Значки (*icon*) — картинки формата *.ico*.
- Картинки (*bitmap*) — картинки формата *.bmp*.
- Курсоры (*cursor*) — картинки формата *.cur*.
- Меню (*menu*) — структуры, описывающие меню.
- Диалоги (*dialog box*) — описания диалоговых окон.
- Строки (*string table*) — таблицы, содержащие строки, например, заголовки окон, надписи на кнопках и т.п.
- Клавиши ускорения (*accelerator table*) — сочетания клавиш, которые генерируют команды, то есть сообщения *WM_COMMAND*.
- Версия (*version information*) — версия программы, и другие.

Пользовательские ресурсы — это любые данные, необходимые для работы программы. Здесь они не описываются.

Ресурсы записываются в текстовый файл с расширением *.rc*. В нем каждому ресурсу сопоставляется идентификатор в виде числа или строки. Для работы с ресурсами можно использовать программу *RC.exe*. В частности, ресурсы можно скомпилировать в двоичный файл с расширением *.res*. Обычно в проект включается файл описания ресурсов *.rc*. При компиляции ресурсов образуется заголовочный файл *resource.h*, который также включается в проект. Этот файл содержит описания идентификаторов ресурсов и используется для извлечения ресурсов и использования их в программе. Если в проект включен файл описания ресурсов *.rc*, на панели *Workspace* появляется вкладка *ResourceView*.

Одним из важнейших ресурсов является значок приложения. Различают стандартный большой значок (*standard*), и маленький (*small*). Стандартный значок имеет размер 32×32 пикселя, маленький — 16×16 пикселей. Заметим, что значок может содержать изображения (*image*) нескольких размеров одновременно. При этом большой значок приложения обычно содержит изображения 32×32 и 16×16, а маленький — только 16×16.

Чтобы изменить количество изображений в значке, следует использовать меню *Image*, которое появляется, когда открыт редактор картинок Visual C++. Для добавления нового изображения нужно выбрать *Image—New Device Image*, и далее — размер изображения и количество цветов. Чтобы удалить изображение из значка, его нужно выбрать в списке *Device* на панели инструментов редактора изображений и далее выбрать в меню *Image—Delete Device Image*.

Ресурс может быть добавлен в проект при помощи меню *Insert—Resource*. Далее следует выбрать тип ресурса и источник. Для создания ресурса средствами Visual C++ следует щелкнуть кнопку *New*, а для добавления готового ресурса следует щелкнуть кнопку *Import*.

Файл описания ресурсов должен быть включен в проект приложения при помощи меню *Project—Add To Project—Files…*. В появившемся диалоге следует найти и выбрать файл типа *.rc*. Заголовочный файл ресурсов *resource.h* включается в проект аналогичным образом. Файл описания ресурсов можно подготовить заранее при помощи Visual C++. Для этого нужно закрыть все проекты, выбрать в меню *File—New*, вкладка *Files*, тип файла — *Resource Script*.

Приложение диалогового типа

Проект

Прежде всего нужно создать подходящий проект.

- Открываем Visual C++.
- Выбираем в меню *File—New*.
- На вкладке *Projects* выбираем *Win32 Application*.
- В поле *Location* вводим путь к каталогу, в котором будет расположен каталог проекта, например C:\. Сам каталог проекта создается автоматически и получает имя проекта.
- В поле *Project name* вводим название проекта, например, *KSGame*.
- Щелкаем кнопку *OK*.
- В диалоге *Step 1 of 1* выбираем *A simple Win32 application*.
- Щелкаем кнопку *Finish*.
- Щелкаем кнопку *OK*.

В результате будет создан проект, состоящий из нескольких файлов, просмотреть которые можно при помощи вкладки *FileView* на панели *Workspace*.

Основной файл проекта — *KSGame.cpp*. Он содержит пустую основную функцию *WinMain*, с которой начинается выполнение программы:

```
int APIENTRY WinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR     lpCmdLine,
                     int       nCmdShow)
{
    // TODO: Place code here.

    return 0;
}
```

Параметрами основной функции являются:

hInstance	— дескриптор приложения — число, указывающее на приложение. На самом деле это адрес размещения кода программы в виртуальном адресном пространстве приложения. Параметр следует запомнить в глобальной переменной.
hPrevInstance	— дескриптор предыдущей копии приложения. Используется для совместимости с Win16. Всегда *NULL*.
lpCmdLine	— хвост командной строки.
nCmdShow	— число, указывающее на начальное состояние главного окна приложения. Оно может быть открыто нормальным образом, развернуто на весь экран, свернуто в значок и т.п.

Основная функция должна инициализировать необходимые данные, создать главное окно и войти в цикл приема и обработки поступающих в приложение сообщений (описывается ниже).

Проект не содержит ресурсов. Поэтому первой задачей является добавление существенно важных ресурсов, прежде всего — значка приложения. Дополнительно следует описать меню и некоторые важные строки.

Ресурсы

К проекту нужно добавить некоторые ресурсы — большой и маленький значки, меню, разные строковые данные и т.п.

Прежде всего добавим в проект файл описания ресурсов:

- Меню *File—New*.
- Вкладка *Files*.
- Выбираем тип файла *Resource Script*.
- В поле *File name* вводим название файла *KSGame*.
- Кнопка *OK*.

В проект будет включен файл *KSGame.rc*, на панели *Workspace* появится вкладка *ResourceView*, и откроется редактор ресурсов, в котором видна папка с названием *KSGame*. Его нужно закрыть *File—Close* или крестиком «Закрыть».

Добавляем в ресурсы значок (ничего не сохраняем до указания):

- Переходим на вкладку *ResourceView*.
- Щелкаем ПРАВОЙ кнопкой мыши на заголовок *KSGame resources*.
- Выбираем в контекстном меню *Insert*. Появится диалог *Insert Resource*, предназначенный для добавления нового ресурса.
- Выбираем тип ресурса *Icon* и щелкаем кнопку *New*. Откроется редактор изображений с открытым пустым (зеленым) значком стандартного размера. Одновременно в списке ресурсов появится группа *Icon*, а в ней — новый ресурс, обозначенный *IDI_ICON1*.
- Сейчас нужно нарисовать значок приложения (рисунок 1, слева). Значок может содержать прозрачные области. В редакторе прозрачный цвет обозначается зеленым и находится в палитре цветов внутри мониторчика.

Добавляем в значок еще одно изображение:

- Меню *Image—New Device Image*.
- Выбираем *Small (16x16)*.
- Кнопка *OK*.
- Рисуем маленький значок (рисунок 2, справа).
- Щелкаем ПРАВОЙ на название ресурса *IDI_ICON1* и выбираем *Properties*.
- Меняем название на *IDI_KSGame*.
- *Enter*.
- Сохраняем *Ctrl+S*.

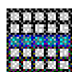

Рисунок 2. Значки приложения

Создаем маленький значок, предназначенный для заголовка окна.

- Щелкаем ПРАВОЙ на заголовок *KSGame resources*.
- Выбираем в контекстном меню *Insert*.
- Выбираем тип ресурса *Icon* и щелкаем кнопку *New*.
- Добавляем маленькое изображение *Image—New Device Image*.
- Выбираем *Small (16x16)*.
- Кнопка *OK*.
- Выбираем *Standard (32x32)* в списке *Device*.

- Удаляем большое изображение *Image—Delete Device Image*.
- Снова рисуем маленький значок или копируем из предыдущего.
- Щелкаем ПРАВОЙ на название ресурса *IDI_ICON1* и выбираем *Properties*.
- Меняем название на *IDI_KSSmall*.
- *Enter*.
- Сохраняем *Ctrl+S*.

Значки готовы. Переходим к созданию ресурса меню.

- Переходим на вкладку *ResourceView*.
- Щелкаем ПРАВОЙ кнопкой мыши на заголовок *KSGame resources*.
- Выбираем в контекстном меню *Insert*.
- Выбираем тип ресурса *Menu* и щелкнем кнопку *New*. Откроется редактор меню. В списке ресурсов появится группа *Menu*, а в ней — ресурс, обозначенный *IDR_MENU1*.
- В редакторе меню щелкаем на пустой пункт меню и вводим название основного пункта *&Игра*. Важно: знак & указывает на символ ускоренного выбора — буква *И* в слове *Игра* будет впоследствии подчеркнута.
- *Enter*.
- Появится подпункт пункта *Игра*, вводим в него *&Закрыть*.
- *Enter*.
- Щелкаем ПРАВОЙ на название ресурса *IDR_MENU1* и выбираем *Properties*.
- Меняем название на *IDR_KSGame*.
- *Enter*.
- Щелкаем на название подпункта *&Закрыть* и выберем *Properties*.
- Заменяем идентификатор подпункта на *IDM_EXIT*.
- *Enter*.
- Сохраняем *Ctrl+S*.

Переходим к таблице строк. Добавим строку — заголовок приложения.

- Щелкаем ПРАВОЙ кнопкой мыши на заголовок *KSGame resources*.
- Выбираем в контекстном меню *Insert*.
- Выбираем тип ресурса *String Table* и щелкаем кнопку *New*. Откроется таблица строк. В списке ресурсов появится группа *String Table*, а в ней — ресурс, обозначенный *String Table*.
- Дважды щелкаем в пустую строку таблицы. Откроется окно свойств.
- Изменяем идентификатор строки на *IDS_APP_TITLE*.
- В поле *Caption* вводим саму строку — *KSGame*.
- *Enter*.
- Сохраняем *Ctrl+S*.

Основные ресурсы добавлены.

Включаем заголовочный файл *resource.h* в проект.

- Меню *Project—Add To Project—Files…*
- Выбираем файл *resource.h*.
- Кнопка *OK*.

Проект готов для создания главного окна приложения. В дальнейшем в него можно будет добавлять другие необходимые ресурсы.

Глобальные объекты

Теперь можно приступить непосредственно к созданию главного окна приложения, однако сначала лучше выполнить некоторые предварительные действия. Сейчас проект должен строиться без ошибок при нажатии клавиши *F7* или при выборе в меню *Build—Build KSGame.exe*. Если проект не строится, нужно найти и устранить все ошибки.

Как говорилось ранее, в основной функции выполняется начальная инициализация программы. Начнем с того, что определим некоторые глобальные константы и переменные.

Добавляем в проект новый заголовочный файл:

- *File—New*.
- Вкладка *Files*.
- Выбираем тип файла *C/C++ Header File*.
- В поле *File name* вводим имя файла *KSGame*.
- *OK*.

В проект добавлен новый файл, он сразу открыт. Защищаем файл от повторного включения при помощи директив условной компиляции:

```
#if !defined(__KSGAME_H__)
#define __KSGAME_H__


#endif
```

Весь код файла должен находиться после директивы #define и перед директивой #endif. Объявляем константы и переменные:

```
// Максимальная длина строки ресурса
#define MAX_STRING 80

// Дескриптор приложения
HINSTANCE hAppInst;
// Заголовок приложения
TCHAR szAppTitle[MAX_STRING];
// Дескриптор главного окна
HWND hWndMain;
```

Закрываем файл *KSGame.h* и открываем *KSGame.cpp*. Включаем заголовочные файлы *KSGame.h* и *resource.h* при помощи директив #include:

```
#include "KSGame.h"
#include "resource.h"
```

В основной функции убираем строчку *TODO* и сохраняем дескриптор:

```
{
    // сохраняем дескриптор приложения
    hAppInst = hInstance;
    return 0;
}
```

Строим проект *F7*. Если есть ошибки, устраняем их.

Следующий шаг — извлекаем строку заголовка из ресурса:

```
{
    // сохраняем дескриптор приложения
    hAppInst = hInstance;
    // извлекаем строку заголовка
    LoadString(hInstance, IDS_APP_TITLE, szAppTitle, MAX_STRING);
    return 0;
}
```

Далее можно создавать главное окно. Перед этим лучше сначала определить его оконную процедуру.

Оконная процедура

Это процедура специального вида (*call-back*), которая обрабатывает сообщения, поступающие в окно из цикла обработки сообщений приложения.

Параметрами оконной процедуры являются дескриптор окна *hWnd*, сообщение *message* и два слова *wParam* и *lParam*, содержащих дополнительную информацию, прикрепляемую к сообщению.

Описываем оконную процедуру *MainProc* после основной функции:

```
// Обработка сообщений главного окна
LRESULT CALLBACK MainProc(HWND hWnd, UINT message, WPARAM wParam,
                                                  LPARAM lParam)
{
    int wmId, wmEvent;
    PAINTSTRUCT ps;
    HDC hdc;
    switch (message) {
    case WM_COMMAND:
        wmId = LOWORD(wParam);
        wmEvent = HIWORD(wParam);
        switch (wmId) {
        case IDM_EXIT:
            DestroyWindow(hWnd);
            return 0;
        }
        break;
    case WM_CREATE:
        // Создание дочерних окон
        break;
    case WM_PAINT:
        hdc = BeginPaint(hWnd, &ps);
        EndPaint(hWnd, &ps);
        return 0;
    case WM_DESTROY:
        PostQuitMessage(0);
        return 0;
    }
    return DefWindowProc(hWnd, message, wParam, lParam);
}
```

Оконная процедура либо обрабатывает сообщение и возвращает ноль, либо передает обработку процедуре по умолчанию *DefWindowProc*. Разбор важнейших сообщений выполняется при помощи оператора *switch*:

WM_COMMAND — команда меню или сообщение о событии элемента управления, не имеющего своей оконной процедуры. Здесь обрабатывается только одна команда, имеющая идентификатор *IDM_EXIT*. Этот идентификатор был определен при создании меню приложения. Как видим, идентификатор команды *wmId* извлекается из дополнительного слова *wParam*. Получив сообщение *IDM_EXIT*, оконная процедура вызывает функцию уничтожения окна *DestroyWindow*. Эта функция посылает сообщение *WM_DESTROY*, которое получает эта же оконная процедура и разбирает его ниже.

WM_CREATE — сообщение, посылаемое при создании окна. Обычно используется для его конструирования, например, для создания дочерних окон, в том числе элементов управления, а также для создания графических объектов.

WM_PAINT — сообщение о необходимости перерисовки клиентской части окна. Актуально для графических приложений. Всё рисование должно происходить после вызова функции *BeginPaint* и перед вызовом функции *EndPaint*.

WM_DESTROY — сообщение об уничтожении окна. Его посылает функция *DestroyWindow*. Реакция на это сообщение заключается в уничтожении разных объектов, например, графических, и посылке сообщения *WM_QUIT*. Это сообщение поступает в основной цикл обработки сообщений приложения и приводит к его завершению. При этом программа выходит из основной функции и завершает существование.

Главное окно

Программист описывает интерфейс, самостоятельно создавая окна при помощи функций *CreateWindow* и *CreateWindowEx*.

Функция *CreateWindowEx*:

```
HWND CreateWindowEx(
    DWORD dwExStyle,      -- расширенный стиль окна
    LPCTSTR lpClassName,  -- зарегистрированное имя класса окна
    LPCTSTR lpWindowName, -- заголовок окна
    DWORD dwStyle,        -- стиль окна
    int x,                -- позиция окна X (внутри другого окна)
    int y,                -- позиция окна Y
    int nWidth,           -- ширина окна
    int nHeight,          -- высота окна
    HWND hWndParent,      -- дескриптор родительского окна
    HMENU hMenu,          -- дескриптор меню или ID дочернего окна
    HINSTANCE hInstance,  -- дескриптор приложения
    LPVOID lpParam        -- дополнительные данные для создания
);
```

У функции *CreateWindow* первый аргумент отсутствует. Обе функции возвращают дескриптор окна или ноль (*NULL*), если окно не создано. Факт создания окна следует проверять.

Имя класса окна — это строка, которая должна быть уникальной для приложения среди всех других классов окон. Название нужно придумать и зареги

стрировать в системе (см. ниже). Следует также знать, что существует несколько предопределенных (зарегистрированных) названий классов, при помощи которых можно создать специфичное окно, обычно элемент управления, такой, как кнопка, список, текст, картинка и т.п., а также дочернее окно MDI. Основные предопределенные классы окон приведены в приложении.

Стиль окна — это сумма констант, определяющая основной вид окна. Константы выбираются из таблицы 1 приложения.

Расширенный стиль окна — это сумма констант, определяющая дополнительные характеристики окна. Константы расширенного стиля приведены в таблице 2 приложения.

Заголовок окна — это то, что появляется в строке заголовка. Если окно не имеет заголовка, то этот параметр может быть принят равным нулю. Если же он указан, то это текст, который появится в окне.

Дескриптор родительского окна задается только для дочерних окон. Для главного окна приложения этот параметр принимается равным нулю.

Дескриптор меню — это идентификатор ресурса типа «меню», которое сопоставляется с окном. Для дочерних окон вместо него может быть задан числовой идентификатор, приведенный к типу *HMENU*, используемый программой по своему усмотрению.

Дополнительные данные для создания — это указатель на структуру, которая описывает данные, необходимые для создания окна. Указатель на эту структуру передается как параметр *lParam* события *WM_CREATE*. Для создания обычного окна используется структура *CREATESTRUCT*, а для создания дочернего окна MDI — структура *CLIENTCREATESTRUCT*.

Главное окно не может иметь предопределенный класс, так как оно должно обладать заголовком. Кроме того, главное окно обязательно должно быть связано с процедурой обработки сообщений. Поэтому создание окна приложения включает в себя еще одну часть, предшествующую вызову функции *CreateWindowEx* — описание характеристик окна и регистрацию.

Для описания свойств класса окна используется структура *WindowClass* или *WindowClassEx*. Рассмотрим вторую структуру.

```
typedef struct _WNDCLASSEX {
    UINT        cbSize;         -- размер структуры
    UINT        style;          -- стиль окна
    WNDPROC     lpfnWndProc;    -- указатель на оконную процедуру
    int         cbClsExtra;     -- дополнительные данные класса
    int         cbWndExtra;     -- дополнительные данные окна
    HINSTANCE   hInstance;      -- дескриптор приложения
    HICON       hIcon;          -- большой значок
    HCURSOR     hCursor;        -- курсор окна
    HBRUSH      hbrBackground;  -- кисть фона окна
    LPCTSTR     lpszMenuName;   -- название ресурса меню окна
    LPCTSTR     lpszClassName;  -- класс окна
    HICON       hIconSm;        -- маленький значок
} WNDCLASSEX, *PWNDCLASSEX;
```

Главное, что есть в этой структуре — это указатель на оконную процедуру *lpfnWndProc*. Заметим, что оконную процедуру можно также установить после создания окна при помощи функции *SetWindowLong*.

Другие поля структуры определяют важнейшие параметры класса окна.

Стиль окна — то же, что и стиль окна в функции *CreateWindowEx*.

Дополнительные данные класса — это одно или несколько чисел типа *long*, которые сопоставляются с классом окна. Для хранения этих чисел выделяется дополнительная память, приписываемая к классу в целом. При помощи системных функций *SetWindowLong* и *GetWindowLong* дополнительные данные могут быть записаны и прочитаны. Максимальный размер дополнительной памяти — 40 байт (10 чисел).

Дополнительные данные окна — то же самое, что и дополнительные данные класса, только память выделяется для каждого представителя класса (для каждого созданного окна), а не для класса в целом.

Курсор окна — это значок указателя мыши, используемый системой, когда мышь находится над областью окна. Обычно это указательная стрелка.

Кисть фона окна — это графический объект «кисть», который используется для закрашивания внутренней (клиентской) части окна. Выбирается системный цвет плюс 1, который приводится к типу *HBRUSH*.

Название ресурса меню окна — это идентификатор (строка), описывающий ресурс типа «меню», который содержит описание меню класса.

После рассмотрения многочисленных параметров приступаем к описанию создания главного окна приложения. Прежде всего перед основной функцией необходимо объявить прототип функции, которая будет создавать главное окно. Это необходимо потому, что функция создания окна будет расположена ниже основной функции:

```
// Прототипы функций модуля
int CreateMainForm(HINSTANCE hInstance, int nCmdShow);
```

Затем ниже оконной процедуры описываем саму функцию создания окна:

```
// Создание главного окна программы
int CreateMainForm(HINSTANCE hInstance, int nCmdShow)
{
    return TRUE;
}
```

Функция создания главного окна начинается с описания переменной типа структуры *WNDCLASSEX* и заполнения её полей типичным образом:

```
    WNDCLASSEX wcex;
    // Обнуляем поля структуры
    memset(&wcex, 0, sizeof(WNDCLASSEX));
    wcex.cbSize         = sizeof(WNDCLASSEX);
    wcex.style          = CS_HREDRAW | CS_VREDRAW;
    wcex.lpfnWndProc    = (WNDPROC)MainProc;
    wcex.hInstance      = hInstance;
    wcex.hIcon          = LoadIcon(hInstance, (LPCTSTR)IDI_KSGame);
    wcex.hCursor        = LoadCursor(NULL, IDC_ARROW);
    wcex.hbrBackground = (HBRUSH)(COLOR_BTNFACE+1);
    wcex.lpszMenuName  = (LPCSTR)IDR_KSGame;
    wcex.lpszClassName = "KSMainWindow";
    wcex.hIconSm        = LoadIcon(hInstance, (LPCTSTR)IDI_KSSmall);
```

Первый оператор (*memset*) обнуляет поля структуры, поэтому некоторые поля можно не заполнять.

После того, как структура *WNDCLASSEX* заполнена необходимыми параметрами, вызываем функцию *RegisterClassEx*, которая регистрирует класс окна в системе. Параметром функции является указатель на структуру:

```
// Регистрация класса главного окна
if (!RegisterClassEx(&wcex))
    return FALSE;
```

Функция возвращает атом в случае успеха или ноль, если окно не зарегистрировано. В случае неуспешной регистрации покидаем функцию с признаком неудачи, при этом приложение должно завершить свою работу.

Наконец, вызываем функцию *CreateWindowEx*.

```
hWndMain = CreateWindowEx(
    WS_EX_CLIENTEDGE, "KSMainWindow", szAppTitle,
    WS_OVERLAPPEDWINDOW,
    100, 100, 512, 354, 0, 0, hInstance, 0);
```

Необходимо проверить, что окно создано:

```
if (!hWndMain)
    return FALSE;
```

Поскольку мы создаем главное окно приложения, дополнительно нужно вызвать функции *ShowWindow* и *UpdateWindow*. Они отображают окно на экране:

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

Далее функция создания окна возвращает истину:

```
return TRUE;
```

Переходим в основную функцию и вызываем функцию создания окна:

```
// Создаем главное окно
if (!CreateMainForm(hInstance, nCmdShow))
    return FALSE;
```

После создания главного окна основная функция переходит в цикл приема и обработки сообщений:

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

Заметим, что основная функция возвращает не ноль, как было вначале, а параметр *wParam* сообщения. Переменную *msg* нужно объявить в начале основной функции:

```
MSG msg;
```

На этом создание основы приложения закончено. Проект нужно построить при помощи *F7* и запустить при помощи *F5*.

Стили окна

Внешний вид окна определяется его стилями, основным и расширенным. Перед продолжением разработки программы необходимо выбрать правильные стили, соответствующие назначению окна (приложения).

Окно, полученное в результате предварительных действий, является окном перекрывающегося типа. У него есть заголовок, меню и границы, которые позволяют произвольно изменять размер окна. Клиентская часть окна имеет утопленную (*sunken*) рамку, и это есть следствие применения расширенного стиля *WS_EX_CLIENTEDGE*.

Для начала посмотрим, как можно изменить данное окно — добавим в него вертикальную и горизонтальную линейки прокрутки. Их наличие определяется константами основного стиля. Изменяем вызов функции *CreateWindowEx*:

```
hWndMain = CreateWindowEx(
    WS_EX_CLIENTEDGE, "KSMainWindow", szAppTitle,
    WS_OVERLAPPEDWINDOW | WS_HSCROLL | WS_VSCROLL,
    100, 100, 512, 354, 0, 0, hInstance, 0);
```

Теперь в клиентской части окна появились линейки прокрутки. В задачу программиста входит управление содержимым окна при событиях линеек прокрутки. Здесь мы это не рассматриваем.

Заменим расширенный стиль *WS_EX_CLIENTEDGE* на ноль. В результате получим окно, клиентская часть которого не утоплена.

Заменим основной стиль на ноль. Получим окно диалогового типа, без системного меню и без границ, которыми можно изменять размер.

Заменим основной стиль на *WS_POPUP*. Получим окно без рамки и без заголовка.

Заменим основной стиль на *WS_POPUPWINDOW*. Получим окно с тонкой рамкой, но без заголовка.

Заменим основной стиль на *WS_CAPTION*. Получим окно диалогового типа без системного меню.

Заменим основной стиль на сочетание *WS_CAPTION* и *WS_SIZEBOX*. Получим странное окно без системного меню, но которое можно изменять в размерах.

В конце концов нужно выяснить, что мы хотим получить. На самом деле мы строим приложение диалогового типа. Следовательно, нам нужно окно диалогового типа (*popup*). Окно не должно изменять свой размер, но должно иметь системное меню и кнопочку «Закрыть» в заголовке. Клиентская часть должна иметь выступ, а не углубление, но пока мы оставим углубление. Получить такое окно можно следующим сочетанием стилей:

```
hWndMain = CreateWindowEx(
    WS_EX_CLIENTEDGE, "KSMainWindow", szAppTitle,
    WS_POPUPWINDOW | WS_CAPTION,
    100, 100, 515, 368, 0, 0, hInstance, 0);
```

Все, что осталось сделать для данного главного окна приложения — установить правильный размер и положение. На данном этапе точный размер неизвестен, поэтому мы отложим это на тот момент, когда окно получит все видимые элементы изображения. Положение также вычислим позже.

Интерфейс

Конечный вид главного окна программы приведен на следующем рисунке.

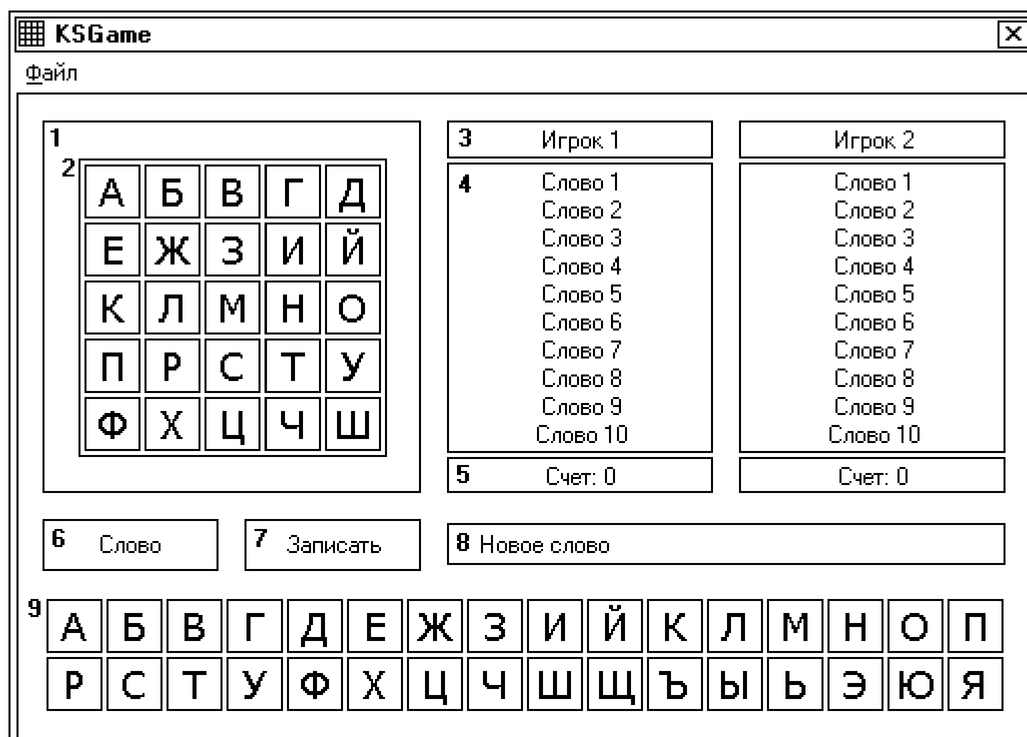

Рисунок 3. Расположение и нумерация элементов интерфейса

Здесь же проставлена нумерация некоторых элементов интерфейса — окон, которые нужно создать.

Для расположения окон нужно определить константы, задающие положение и размеры окон, а также идентификаторы кнопок. Модуль *KSGame.h*.

```
// Размеры клетки
#define CLW 30
#define CLH 29
#define CLB 2
// Пробел по ширине
#define SPW 12
// Пробел по высоте
#define SPH 12
// Высота элемента списка
#define PHI 14
// Рамка игрового поля
#define PW1 190
#define PH1 187
#define PX1 SPW
#define PY1 SPH
// Игровое поле
#define PW2 CLW*5+CLB
#define PH2 CLH*5+CLB
#define PX2 (PW1-4-PW2)/2
#define PY2 (PH1-4-PH2)/2
```

```
// Рамка имени игрока
#define PW3 134
#define PH3 20
#define PX3 PX1+PW1+SPW
#define PY3 PY1
// Рамка списка слов
#define PW4 PW3
#define PH4 146
#define PX4 PX3
#define PY4 PY3+PH3+1
// Рамка счета игрока
#define PW5 PW3
#define PH5 19
#define PX5 PX3
#define PY5 PY4+PH4+1
// Кнопка "Слово"
#define PW6 88
#define PH6 26
#define PX6 PX1
#define PY6 PY1+PH1+SPH
// Кнопка "Записать"
#define PW7 PW6
#define PH7 PH6
#define PX7 PX6+PW6+SPW+1
#define PY7 PY6
// Рамка нового слова
#define PW8 PW3+PW3+SPW
#define PH8 PH6-4
#define PX8 PX3
#define PY8 PY6+2
// Алфавит
#define PW9 CLW*16+CLB
#define PH9 CLH*2+CLB
#define PY9 PY6+PH6+SPH
#define PX9 PX1
// Идентификатор кнопки "Слово"
#define IDB_WORD 301
// Идентификатор кнопки "Записать"
#define IDB_WRITE 302
```

Кроме того, нужны переменные для хранения дескрипторов, а также контексты окон и идентификаторы кнопок:

```
// Дескрипторы
// Клетки
HWND hWndCell[25];
// Окна букв алфавита
HWND hWndLetter[32];
// Окна имен игроков
HWND hWndName[2];
// Окна счетов игроков
HWND hWndCount[2];
```

```
// Окна списков игроков
HWND hWndList[2][10];
// Окно нового слова
HWND hWndWord;
// Контексты
// Контексты клеток
HDC hdcCell[25];
// Контексты букв алфавита
HDC hdcLetter[32];
```

Дополнительно нужно хранить надписи на клетках, буквы алфавита, состояния клеток и букв, счета игроков и счетчики списков:

```
// Буквы клеток
TCHAR CellText[25];
// Буквы алфавита
TCHAR LetterText[] = "АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ";
// Выделения клеток
int CellSel[25];
// Выделения букв алфавита
int LetterSel[32];
// Имена игроков
TCHAR szPlayerName[2][MAX_STRING];
// Строки счетов игроков
TCHAR szPlayerCount[2][20];
// Счета игроков
int PlayerCount[2] = { 0, 0 };
// Счетчики списков
int lcount[2] = { 0, 0 };
```

Кроме того, нужны графические объекты — кисти и шрифты и цвета:

```
// Кисти
HBRUSH DefBrush, SelBrush;
// Шрифты
HFONT MainFont, CellFont;
// Цвета
COLORREF DefBackColor = GetSysColor(COLOR_BTNFACE);
COLORREF SelBackColor = RGB(0, 0, 200);
COLORREF DefForeColor = RGB(0, 0, 0);
COLORREF SelForeColor = RGB(255, 255, 255);
```

Дочерние окна

Дочерние окна должны определять стиль *WS_CHILD*. Это окна-контейнеры и элементы управления. Они находятся внутри других (родительских) окон. Назначение контейнера — объединить другие дочерние окна (например, элементы управления). Дочерние окна обычно создаются при получении сообщения *WM_CREATE* в оконной процедуре главного (родительского) окна. Однако их можно создавать и вне связи с этим сообщением, но на момент создания дочернего окна нужно иметь дескриптор родительского окна.

Первое дочернее окно в нашем приложении будет создавать рамку игрового поля и являться контейнером для окна игрового поля. В принципе, это окно

— просто некоторая рамка. Подойдет наиболее простой предопределенный класс окна — это класс *STATIC*. Поскольку это окно будет являться родительским для окна игрового поля, понадобится его дескриптор *W1*, а также дополнительный дескриптор *W2*. Это локальные переменные оконной процедуры:

```
LRESULT CALLBACK MainProc(HWND hWnd, UINT message, . . .
{
    HWND W1, W2;
```

Окно создается в обработке события *WM_CREATE*. Сейчас эта обработка заканчивается оператором *break*, который передает управление процедуре обработки по умолчанию. Оператор *break* нужно заменить на оператор *return 0*.

Поскольку окно не имеет оконной процедуры, его можно создать простым вызовом функции *CreateWindow* (не *CreateWindowEx*):

```
    case WM_CREATE:
        // Рамка игрового поля
        W1 = CreateWindow("STATIC", 0,
            WS_CHILD | WS_VISIBLE | SS_ETCHEDFRAME,
            PX1, PY1, PW1, PH1, hWnd, 0, hAppInst, 0);
        return 0;
```

Константа стиля *SS_ETCHEDFRAME* создает оригинальную рамку.

Следующее окно определяет фон игрового поля (белый). В этом окне будут располагаться клетки игрового поля:

```
        // Окно игрового поля
        W2 = CreateWindow("STATIC", 0,
            WS_CHILD | WS_VISIBLE | SS_WHITERECT,
            PX2, PY2, PW2, PH2, W1, 0, hAppInst, 0);
```

Обратим внимание, что дескриптор *W1* используется как дескриптор родительского окна для окна игрового поля. Поэтому новое окно размещается внутри предыдущего. Стиль *SS_WHITERECT* создает окно с белым фоном.

Дальше нужно создать 25 окон — клетки игрового поля. Эти окна нужно упорядоченно разместить внутри окна игрового поля. Эти окна важны для программы в том смысле, что нам нужно отслеживать действия игрока с клетками поля. Поэтому окна клеток игрового поля должны иметь оконные процедуры.

Кроме того, клетки поля постоянно должны перекрашиваться. Закрашивание окна связано с получением его контекста. Мы создали массивы для сохранения контекстов и дескрипторов окон клеток и описали кисти. Перед созданием окна рамки игрового поля теперь создаём кисти:

```
    case WM_CREATE:
        // Создаём кисти
        DefBrush = CreateSolidBrush(DefBackColor);
        SelBrush = CreateSolidBrush(SelBackColor);
```

Кисти нужно уничтожить в событии *WM_DESTROY*:

```
    case WM_DESTROY:
        // Уничтожаем кисти
        DeleteObject(DefBrush);
        DeleteObject(SelBrush);
```

Перед тем, как создавать окна клеток, опишем оконную процедуру. Располагаем её *перед* оконной процедурой главного окна:

```
// Обработка сообщений окна клетки
LRESULT CALLBACK CellProc(HWND hWnd, UINT message, WPARAM wParam,
                                                    LPARAM lParam)
{
    PAINTSTRUCT ps;
    switch (message) {
    case WM_LBUTTONDOWN:
    case WM_LBUTTONDBLCLK:
        // На клетку щелкнули
        return 0;
    case WM_LBUTTONUP:
        // Мышь отпустили
        return 0;
    case WM_PAINT:
        // Требуется перерисовка клетки
        BeginPaint(hWnd, &ps);
        EndPaint(hWnd, &ps);
        return 0;
    }
    // Другие сообщения обрабатывает класс STATIC
    return DefWindowProc(hWnd, message, wParam, lParam);
}
```

Обработку сообщений окна клетки мы опишем позднее. В оконной процедуре главного окна после создания окна игрового поля описываем создание окон клеток. После создания каждой клетки устанавливаем дополнительные параметры, такие, как оконную процедуру, прозрачность текста, начальный текст клетки и состояние:

```
        // Создаём окна клеток
        for (wmId = 0; wmId < 25; wmId++) {
            hWndCell[wmId] = CreateWindow("STATIC", 0,
                WS_CHILD | WS_VISIBLE,
                CLB + CLW * (wmId % 5),
                CLB + CLH * (wmId / 5),
                CLW - CLB, CLH - CLB,
                W2, (HMENU)wmId, hAppInst, 0);
            // Прикрепляем оконную процедуру
            SetWindowLong(hWndCell[wmId],
                GWL_WNDPROC, (LONG)CellProc);
            // Получаем контекст окна клетки
            hdcCell[wmId] = GetDC(hWndCell[wmId]);
            // Устанавливаем прозрачность текста
            SetBkMode(hdcCell[wmId], TRANSPARENT);
            // Начальный текст клетки
            CellText[wmId] = 'A' + wmId; // !!! требуется пробел
            // Клетка изначально не выделена
            CellSel[wmId] = 0;
        }
```

Обратим внимание, что при создании окон вместо идентификатора меню мы записали переменную *wmId*. Она служит числовым идентификатором окна и является аналогом свойства *Index* в языке Visual Basic. Индекс позволит впоследствии определить, какая клетка обрабатывается в оконной процедуре.

Поскольку при создании окон клеток мы создавали контексты, требуется их освобождение при разрушении окна. Поэтому переходим в обработку события *WM_DESTROY* и после уничтожения кистей освобождаем контексты:

```
// Уничтожаем контексты
for (wmId = 0; wmId < 25; wmId++) {
    ReleaseDC(hWndCell[wmId], hdcCell[wmId]);
}
```

Несмотря на то, что окна созданы, мы не увидим их, так как они не закрашиваются — в оконной процедуре в перерисовке клеток пусто. Поэтому следующая задача — разработать процедуру для закраски клетки *DrawCell*.

Объявляем процедуру закраски в заголовочном модуле *KSGame.h*. Параметр *index* этой процедуры указывает на конкретную клетку, а параметр *select* — на её новое состояние, которое следует запомнить:

```
// Рисует состояние клетки
void DrawCell(int index, int select)
{
}
```

Сначала в процедуре объявляем структуру *Rect* для получения прямоугольника клиентской части окна и получаем его:

```
RECT rc;
// Получаем размер клиентской части окна
GetClientRect(hWndCell[index], &rc);
```

Далее в зависимости от того, нужно клетку выделить, или убрать выделение, закрашиваем её соответствующей кистью и устанавливаем цвет текста:

```
if (select == 0) {
    // Закрашиваем основной кистью
    FillRect(hdcCell[index], &rc, DefBrush);
    // Устанавливаем основной цвет текста
    SetTextColor(hdcCell[index], DefForeColor);
} else {
    // Закрашиваем выделяющей кистью
    FillRect(hdcCell[index], &rc, SelBrush);
    // Устанавливаем выделенный цвет текста
    SetTextColor(hdcCell[index], SelForeColor);
}
```

В заключение выводим текст и запоминаем новое состояние клетки:

```
// Выводим одну букву в центре окна
DrawText(hdcCell[index], &CellText[index], 1, &rc,
    DT_CENTER | DT_VCENTER | DT_SINGLELINE);
// Устанавливаем новое состояние
CellSel[index] = select;
```

Чтобы текст располагался в окне по центру, в функции *DrawText* мы выбрали подходящие константы выравнивания (последний аргумент). При этом константу *DT_SINGLELINE* требует константа *DT_VCENTER*. Пока мы не затрагиваем вопрос о шрифте, разберемся с ним позднее.

Теперь мы можем показать клетки, вызвав процедуру закраски в обработке события *WM_PAINT* оконной процедуры клетки. Сначала нужно получить индекс клетки (в начале процедуры):

```
// Индекс клетки
int index = GetWindowLong(hWnd, GWL_ID);
```

Собственно закраска заключается в вызове функции *DrawCell*:

```
case WM_PAINT:
    // Требуется перерисовка клетки
    BeginPaint(hWnd, &ps);
    DrawCell(index, CellSel[index]);
    EndPaint(hWnd, &ps);
    return 0;
```

После построения и запуска программы мы должны увидеть игровое поле, заполненное буквами алфавита. Буквы нужны только для отладки. После завершения разработки программы их нужно будет заменить пробелами.

В заключение заметим, что перерисовка клетки (сообщение *WM_PAINT*) возникает в случае, если клетка была закрыта чем-нибудь и затем вновь открыта, например, вследствие перемещения окна приложения.

События мыши

В момент, когда нажимается левая кнопка мыши, система формирует сообщение *WM_LBUTTONDOWN*. Когда левая кнопка мыши отпускается, система формирует сообщение *WM_LBUTTONUP*. Система посылает сообщение *оконной процедуре того окна*, над которым находится курсор мыши в момент события. Иначе говоря, если кнопка мыши была нажата над одним окном, а отпущена над другим, то сообщения посылаются *разным окнам!!!* Это очень важный факт, который следует осознать. В языке Visual Basic, например, событие отпускания кнопки мыши всегда поступает в то окно, над которым кнопка сначала была нажата.

К сообщению обычно прикрепляется дополнительная информация.

Так, при формировании сообщения *WM_LBUTTONDOWN* в дополнительное слово *wParam* записывается число, указывающее на состояние клавиш *Ctrl* и *Shift*, а также на состояние кнопок мыши. Это число является суммой констант, приведенных в таблице 4 приложения.

В дополнительное слово *lParam* записываются координаты курсора: в младшую часть слова — координата *X*, в старшую часть — координата *Y*. При необходимости эта информация может быть извлечена при помощи следующих макросов:

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

Аналогично формируются и другие сообщения (события). В задачу программиста входит обработка сообщений в оконной процедуре.

Для завершения разработки игрового поля в оконной процедуре окна клетки нужно обработать события мыши, такие, как нажатие, двойное (повторное) нажатие, и отпускание кнопки мыши.

Прежде всего нужно понять, что мы хотим получить. А нам хочется, чтобы факт нажатия кнопки мыши над клеткой как-то отображался. Например, при нажатии кнопки клетка меняет свое состояние на противоположное, а при отпускании возвращает предыдущее состояние, то есть снова меняет состояние на противоположное.

Поменять состояние клетки несложно. Нужно вызвать процедуру закраски клетки с инвертированным значением её состояния:

```
case WM_LBUTTONDOWN:
case WM_LBUTTONDBLCLK:
    // На клетку щелкнули
    DrawCell(index, !CellSel[index]);
    return 0;
case WM_LBUTTONUP:
    // Мышь отпустили
    DrawCell(index, !CellSel[index]);
    return 0;
```

Если построить и запустить программу, то при беглом тестировании кажется, что все работает так, как и предполагалось. Но следует быть очень внимательным и проверить, что получится, если нажать кнопку мыши над одной клеткой, а отпустить над другой. Обе клетки окажутся инвертированными. Можно также нажать кнопку мыши над белой границей между клетками (над окном игрового поля), а отпустить над клеткой. Клетка окажется инвертированной. Вряд ли стоит рассчитывать на то, что пользователь будет четко фиксировать точки нажатия и отпускания кнопок мыши, обычно все происходит очень быстро, при одновременном движении мыши, и точка нажатия редко совпадает с точкой отпускания. Поэтому мы *обязаны отслеживать*, в каком окне кнопка мыши была нажата, а в каком отпущена.

К счастью, система предоставляет нам функцию *SetCapture* для захвата событий мыши. Захват (*capture*) гарантирует, что все последующие события мыши будут поступать только в то окно, которое осуществило захват. В конце концов, после получения требуемого сообщения, окно должно «отпустить» захват при помощи процедуры *ReleaseCapture*. Новая версия обработки событий:

```
case WM_LBUTTONDOWN:
case WM_LBUTTONDBLCLK:
    // На клетку щелкнули
    DrawCell(index, !CellSel[index]);
    // Захватываем события мыши
    SetCapture(hWnd);
    return 0;
case WM_LBUTTONUP:
    // Мышь отпустили
    DrawCell(index, !CellSel[index]);
    // Освобождаем захват событий мыши
    ReleaseCapture();
    return 0;
```

Тестируем программу, и убеждаемся, что отпускание кнопки мыши над произвольной клеткой, да и вообще над произвольным местом, инвертирует ту клетку, над которой кнопка была нажата. Однако, если нажать кнопку над белой границей между клетками, а отпустить над какой-нибудь клеткой, клетка инвертируется. Это неправильно, так как приводит к непредусмотренному изменению состояния клетки.

Анализируя ситуацию, начинаем понимать, что при нажатии кнопки не над клеткой захвата не происходит, а поскольку все окна клеток используют одну и ту же оконную процедуру, событие отпускания кнопки приходит в нее и инвертирует какую-то клетку в обработке сообщения *WM_LBUTTONUP*, хотя при этом сообщение *WM_LBUTTONDOWN* в процедуру не поступало.

Следовательно, прежде, чем инвертировать клетку в обработке сообщения *WM_LBUTTONUP*, нужно убедиться в том, что был осуществлен захват событий мыши именно для этой клетки.

Функция *GetCapture* возвращает дескриптор того окна, которое осуществило захват (или ноль, если захвата не было). Если сравнить его с дескриптором окна, которое получило сообщение *WM_LBUTTONUP*, то можно проверить факт нажатия и отпускания кнопки над одним и тем же окном. Изменения касаются только обработки сообщения отпускания кнопки:

```
case WM_LBUTTONUP:
    // Мышь отпустили
    // проверяем, был ли захват мыши
    if (GetCapture() != hWnd)
        break;
    DrawCell(index, !CellSel[index]);
    // Освобождаем захват событий мыши
    ReleaseCapture();
    return 0;
```

Здесь, если дескрипторы окон не совпадают, управление переходит к процедуре обработки сообщений по умолчанию при помощи оператора *break*. Тщательное тестирование программы убеждает нас в ее правильной работе.

Чтобы тестирование было полным, инвертируем одну из клеток после создания окон клеток, например, клетку 11:

```
DrawCell(11, 1);
```

После проверки удалим этот оператор.

В заключение несколько слов о событии «двойной щелчок». Двойной щелчок фиксируется, когда кнопка мыши была повторно нажата над одним и тем же местом в течение наперёд заданного интервала времени, по умолчанию равного 500 мс (полсекунды). Допускается смещение точки нажатия на 1 пиксель. Если повторное нажатие в заданном интервале времени произошло в другой точке, фиксируется обычное нажатие. При повторном нажатии система формирует сообщение *WM_LBUTTONDBLCLK*. Оно посылается тому окну, над которым произошло первое нажатие, или окну, которое осуществило захват.

Последовательность сообщений, которые посылает система при «двойном щелчке», следующая: *WM_LBUTTONDOWN*, *WM_LBUTTONUP*, *WM_LBUTTONDBLCLK*, *WM_LBUTTONUP*.

Заметим, что «двойной щелчок» фиксируется только для левой кнопки.

Шрифты

Шрифт — это графический объект. Шрифт создается одной из функций *CreateFont*, *CreateFontIndirect* или *CreateFontIndirectEx*. Две последних функции требуют заполнения структуры *LogFont* (описывающей параметры логического шрифта), а первая функция определяет все параметры через свои аргументы.

У шрифта множество параметров, но в большинстве случаев достаточно указать только размер (первый аргумент функции *CreateFont*), жирность (пятый аргумент), наклон (курсив, шестой аргумент), набор символов (девятый аргумент) и гарнитуру (название шрифта, последний, четырнадцатый аргумент).

После того, как шрифт создан, его нужно выбрать в контекст устройства при помощи функции *SelectObject*. После окончания использования шрифт нужно удалить при помощи функции *DeleteObject*.

Создадим два шрифта — *MainFont* для надписей и *CellFont* для клеток:

```
case WM_CREATE:
    // Получаем контекст
    hdc = GetDC(hWnd);
    // Разрешение контекста по вертикали
    wmId = GetDeviceCaps(hdc, LOGPIXELSY);
    // Создаем шрифт клетки
    CellFont = CreateFont(-MulDiv(14, wmId, 72),0,0,0,0,
        0,0,0,RUSSIAN_CHARSET,0,0,0,0,"Verdana");
    // Создаем шрифт надписей
    MainFont = CreateFont(-MulDiv(8, wmId, 72),0,0,0,0,
        0,0,0,RUSSIAN_CHARSET,0,0,0,0,"MS Sans Serif");
    // Уничтожаем контекст
    ReleaseDC(hWnd, hdc);
```

Сначала при помощи функции *GetDC* мы получаем контекст главного окна. При помощи функции *GetDeviceCaps* вычисляем разрешение экрана по вертикали в точках на дюйм, обычно равное 96 *dpi*. Далее создаем шрифт.

Логический размер шрифта вычисляет функция *MulDiv*, которая умножает требуемый размер 8 или 14 на разрешение экрана и делит на 72 (число пунктов в дюйме). Результат функция *MulDiv* округляет, полученное значение сравнивается с имеющимися размерами шрифта и выбирается ближайшее большее. Знак минус предписывает сравнивать размер с высотой букв шрифта.

В принципе, размер шрифта — это некоторое число (как правило, немного большее требуемого), которое можно подобрать. Однако при изменении разрешения экрана (а в настоящее время имеется тенденция к его увеличению до величины 120 *dpi* и больше), шрифт окажется неподходящим.

Константа *RUSSIAN_CHARSET* определяет набор символов кириллицы. Название шрифта выбирается из тех, которые есть в системе. Вообще говоря, предварительно следовало бы убедиться в наличии требуемой гарнитуры при помощи функции перечисления шрифтов. При необходимости шрифт можно было бы установить из файла шрифта.

После получения шрифтов контекст устройства уничтожается при помощи функции *ReleaseDC*. Дальше шрифт нужно выбрать в контекст устройства для того, чтобы вывод текста происходил с заданным шрифтом. Но сначала несколько слов о контексте *OWNDC*.

OWNDC

Проведем небольшой эксперимент. Выберем шрифт *MainFont* в контекст главного окна и выведем в окно простую надпись.

Добавим оператор выбора шрифта в контекст перед оператором уничтожением контекста:

```
SelectObject(hdc, MainFont);
// Уничтожаем контекст
ReleaseDC(hWnd, hdc);
```

В обработке сообщения *WM_PAINT* выведем в окно надпись:

```
case WM_PAINT:
    hdc = BeginPaint(hWnd, &ps);
    TextOut(hdc, 20, 230, "Закрыть", 7);
    EndPaint(hWnd, &ps);
    return 0;
```

Строим и запускаем приложение. Надпись появляется в окне, но она имеет другой шрифт (полужирный). Нужно разобраться, почему.

А дело в том, что шрифт выбран в одном контексте, который сразу же после этого уничтожается, а надпись выводится в другом. Можно попробовать вывести надпись непосредственно после выбора шрифта в контекст, но тогда мы ее вообще не увидим. Это связано с тем, что в момент вывода надписи окно еще не появилось на экране.

И вообще, если что-то должно быть нарисовано в окне, то оно должно быть нарисовано в обработке сообщения *WM_PAINT*. Это сообщение система как раз и посылает тогда, когда окно нужно нарисовать. Любой графический вывод вне этого события, если и будет появляться в окне, то ненадолго. Любое изображение, расположенное над окном, сотрет все, что было под ним нарисовано.

Поэтому единственный выход сейчас — перенести выбор шрифта в обработку сообщения *WM_PAINT*.

Это, однако, не всегда удобно. Рассмотрим окна клеток. В них рисование происходит в разных событиях. Правда, мы схитрили, и сохранили для них контекст устройства, который используем *и в событии PAINT*. Но если бы мы в событии *PAINT* использовали контекст, возвращаемый функцией *BeginPaint*, то получили бы два разных контекста и два разных эффекта.

Иногда важно, чтобы окно не теряло контекст, а сохраняло его все время существования. Для этой цели существует специальный стиль окна, обозначаемый константой *CS_OWNDC*. *Own* означает «собственный». Окно, созданной с данным стилем, обладает *собственным контекстом*, и любая функция, которая возвращает контекст устройства, будет возвращать один и тот же контекст, и все настройки, которые для него были сделаны ранее, сохраняются.

Перейдем в функцию создания главного окна и добавим стиль *CS_OWNDC*:

```
wcex.style          = CS_HREDRAW | CS_VREDRAW | CS_OWNDC;
```

Теперь главное окно обладает собственным контекстом, и всегда будет выводить надписи шрифтом, который может быть установлен в любом месте, где можно получить контекст.

На самом деле собственный контекст главного окна нам не нужен, поэтому следует его убрать, а также удалить операторы выбора шрифта и вывода надписи. Все рассуждения были сделаны для прояснения ситуации. Как было сказано, мы схитрили и сохранили контексты клеток, поэтому для завершения работы над клетками нам осталось выбрать шрифт *CellFont* в контекст. Сделать это можно во время создания окна клетки, внутри цикла:

```
        CellSel[wmId] = 0;
        // Выбираем шрифт клетки в контекст
        SelectObject(hdcCell[wmId], CellFont);
    }
```

Шрифты, являясь графическими объектами, располагаются в системной памяти. Их нужно удалять. Переходим в обработку сообщения *WM_DESTROY* главного окна и удаляем созданные шрифты:

```
    // Уничтожаем шрифты
    DeleteObject(MainFont);
    DeleteObject(CellFont);
```

В заключение нужно упомянуть о сообщении *WM_SETFONT*. Программист может послать его любому окну при помощи функции *SendMessage* для того, чтобы уведомить его о необходимости изменения шрифта. Объект «шрифт» при этом посылается как параметр *wParam*. Изменить текст окна можно при помощью функции *SetWindowText*.

Надписи

Надпись — это окно, которое выводит на экран какой-нибудь текст. В принципе текст в окно можно вывести при помощи одной из системных функций. Однако часто требуется менять текст по ходу выполнения программы. Кроме того, иногда требуется получать от текста действия пользователя, такие, как щелчки мышью. Реализовать это можно, если выводить текст внутри специального создаваемого окна предопределенного класса *STATIC*.

В нашей программе несколько надписей. Это имена игроков, счета игроков, а также слово, которое набирает игрок. Кроме того, списки слов игроков мы также реализуем как массивы надписей. Для каждой надписи нам потребуется отдельное окно.

Начинаем создание и расположение надписей с имен игроков. Поскольку данные надписи по творческому замыслу должны быть заключены в рамки, потребуется создать по два окна на каждую надпись.

Поскольку игроков два, создавать надписи будем внутри цикла. Первыми создаем окна-рамки:

```
    // Надписи
    for (wmId = 0; wmId < 2; wmId++) {
        // Рамки окон надписей имен игроков
        W1 = CreateWindow("STATIC", 0,
            WS_CHILD | WS_VISIBLE | SS_ETCHEDFRAME,
            PX3 + (PW3 + PX1) * wmId, PY3, PW3, PH3,
            hWnd, 0, hAppInst, 0);
    }
```

Далее нужно создать надписи имен игроков. Прежде следует создать сами имена. Сделать это можно в главной процедуре *WinMain* после получения строки из ресурса:

```
strcpy(szPlayerName[0], "Игрок 1");
strcpy(szPlayerName[1], "Игрок 2");
```

Внутри цикла создаем надписи имен игроков:

```
        // Окна надписей имен игроков
        hWndName[wmId] = CreateWindow(
            "STATIC", szPlayerName[wmId],
            WS_CHILD | WS_VISIBLE | SS_CENTER,
            2, 3, PW3 - 4, PH3 - 5, W1, 0, hAppInst, 0);
```

После создания надписи посылаем сообщение об изменении шрифта:

```
        SendMessage(hWndName[wmId],
            WM_SETFONT, (WPARAM)MainFont, 0);
```

Тестируем и убеждаемся, что надписи выводятся.

Переходим к созданию списков слов. Списки должны быть заключены в рамку, поэтому начинаем с создания окон рамок:

```
        // Списки
        // Окна рамок
        W1 = CreateWindow("STATIC", 0,
            WS_CHILD | WS_VISIBLE | SS_ETCHEDFRAME,
            PX4 + (PW4 + PX1) * wmId, PY4, PW4, PH4,
            hWnd, 0, hAppInst, 0);
```

Собственно элементы списка создаются внутри еще одного цикла:

```
        // Окна элементов списка
        for (wmEvent = 0; wmEvent < 10; wmEvent++) {
            hWndList[wmId][wmEvent] = CreateWindow(
                "STATIC", 0,
                WS_CHILD | WS_VISIBLE | SS_CENTER,
                2, 3 + PHI * wmEvent, PW4 - 4, PHI,
                W1, 0, hAppInst, 0);
            SendMessage(hWndList[wmId][wmEvent],
                WM_SETFONT, (WPARAM)MainFont, 0);
        }
```

Для управления списками нужны две процедуры. Первая процедура добавляет новое слово в список указанного игрока. Описываем её в заголовочном модуле *KSGame.h*:

```
// Добавляет новый элемент в список игрока
void AddToList(int index, const char* item)
{
    if (lcount[index] < 10) {
        SetWindowText(hWndList[index][lcount[index]], item);
        lcount[index]++;
    }
}
```

Процедура использует функцию *SetWindowText*, которая изменяет текст окна. Вторая процедура очищает указанный список:

```
// Очищает список игрока
void ClearList(int index)
{
    for (int i = 0; i < 10; i++) {
        SetWindowText(hWndList[index][i], "");
    }
    lcount[index] = 0;
}
```

Массив *lcount* хранит количества слов в списках. Проверяем, как работает процедура добавления слова, вызывая её после создания элементов списка:

```
AddToList(wmId, "Слово 1");     // Для отладки
AddToList(wmId, "Слово 2");     // Для отладки
. . .
AddToList(wmId, "Слово 9");     // Для отладки
AddToList(wmId, "Слово 10");    // Для отладки
```

Перед тем, как создать надписи для счетов, в модуле *KSGame.h* описываем вычисление строки счета при помощи функции *StrCount*:

```
// Формирует строку счета игрока
char* StrCount(int index, int count)
{
    // Формируем строку "Счет: "
    strcpy(szPlayerCount[index], "Счет: ");
    // вычисляем счет и записываем в строку вывода
    itoa(count, &szPlayerCount[index][6], 10);
    // возвращаем указатель на строку
    return szPlayerCount[index];
}
```

В модуль *KSGame.h* должен быть включен заголовочный файл *stdlib.h*. Наконец, создаем надписи счетов (все еще внутри цикла *wmId*):

```
            // Рамки окон надписей счетов игроков
            W1 = CreateWindow("STATIC", 0,
                WS_CHILD | WS_VISIBLE | SS_ETCHEDFRAME,
                PX5 + (PW5 + PX1) * wmId, PY5, PW5, PH5,
                hWnd, 0, hAppInst, 0);
            // Окна надписей счетов игроков
            hWndCount[wmId] = CreateWindow(
                "STATIC", StrCount(wmId, 0),
                WS_CHILD | WS_VISIBLE | SS_CENTER,
                2, 3, PW5 - 4, PH5 - 5, W1, 0, hAppInst, 0);
            SendMessage(hWndCount[wmId],
                WM_SETFONT, (WPARAM)MainFont, 0);
        }
```

На этом цикл завершается. Далее создаются окна, относящиеся к алфавиту. Окна букв алфавита ничем не отличаются от клеток игрового поля, поэтому код приводится без пояснений:

```
        // Алфавит
        // Окно алфавита
        W2 = CreateWindow("STATIC", 0,
            WS_CHILD | WS_VISIBLE | SS_WHITERECT,
            PX9, PY9, PW9, PH9, hWnd, 0, hAppInst, 0);
        // Окна букв алфавита
        for (wmId = 0; wmId < 32; wmId++) {
            hWndLetter[wmId] = CreateWindow("STATIC", 0,
                WS_CHILD | WS_VISIBLE,
                CLB + CLW * (wmId % 16),
                CLB + CLH * (wmId / 16),
                CLW - CLB, CLH - CLB,
                W2, (HMENU)(wmId), hAppInst, 0);
            // Прикрепляем оконную процедуру
            SetWindowLong(hWndLetter[wmId],
                GWL_WNDPROC, (LONG)LetterProc);
            // Получаем контекст окна
            hdcLetter[wmId] = GetDC(hWndLetter[wmId]);
            // Устанавливаем прозрачность текста
            SetBkMode(hdcLetter[wmId], TRANSPARENT);
            // Буква изначально не выделена
            LetterSel[wmId] = 0;
            // Выбираем шрифт буквы в контекст
            SelectObject(hdcLetter[wmId], CellFont);
        }
        LetterSel[0] = 1;
```

Процедура рисования буквы алфавита ничем не отличается от процедуры рисования клетки игрового поля. Описываем ее в модуле *KSGame.h*:

```
// Рисует состояние буквы
void DrawLetter(int index, int select)
{
    RECT rc;
    // Получаем размер клиентской части окна
    GetClientRect(hWndLetter[index], &rc);
    if (select == 0) {
        // Закрашиваем основной кистью
        FillRect(hdcLetter[index], &rc, DefBrush);
        // Устанавливаем основной цвет текста
        SetTextColor(hdcLetter[index], DefForeColor);
    } else {
        // Закрашиваем выделяющей кистью
        FillRect(hdcLetter[index], &rc, SelBrush);
        // Устанавливаем выделенный цвет текста
        SetTextColor(hdcLetter[index], SelForeColor);
    }
    // Выводим одну букву в центре окна
    DrawText(hdcLetter[index], &LetterText[index], 1, &rc,
        DT_CENTER | DT_VCENTER | DT_SINGLELINE);
    // Устанавливаем новое состояние
    LetterSel[index] = select;
}
```

Различия проявляются только в оконной процедуре, которую можно расположить после оконной процедуры клеток игрового поля. Она реализует иной алгоритм нажатия кнопок, — нажатая кнопка остается выделенной:

```
// Обработка сообщений окна буквы
LRESULT CALLBACK LetterProc(HWND hWnd, UINT message, WPARAM wParam,
                                                     LPARAM lParam)
{
    PAINTSTRUCT ps;
    // Индекс клетки
    int i, index = GetWindowLong(hWnd, GWL_ID);
    switch (message) {
    case WM_LBUTTONDOWN:
    case WM_LBUTTONDBLCLK:
        // На клетку щелкнули
        if (LetterSel[index] == 0) {
            // Убираем существующее выделение
            for (i = 0; i < 32; i++) {
                if (LetterSel[i] != 0) {
                    DrawLetter(i, 0);
                }
            }
            DrawLetter(index, !LetterSel[index]);
        } else {
            DrawLetter(index, !LetterSel[index]);
            // Захватываем события мыши
            SetCapture(hWnd);
        }
        return 0;
    case WM_LBUTTONUP:
        // Мышь отпустили
        // проверяем, был ли захват мыши
        if (GetCapture() != hWnd)
            break;
        DrawLetter(index, !LetterSel[index]);
        // Освобождаем захват событий мыши
        ReleaseCapture();
        return 0;
    case WM_PAINT:
        // Требуется перерисовка клетки
        BeginPaint(hWnd, &ps);
        DrawLetter(index, LetterSel[index]);
        EndPaint(hWnd, &ps);
        return 0;
    }
    // Другие сообщения обрабатывает класс STATIC
    return DefWindowProc(hWnd, message, wParam, lParam);
}
```

Как и для клеток игрового поля, в обработке события *WM_DESTROY* необходимо уничтожить контексты окон букв алфавита:

```
for (wmId = 0; wmId < 32; wmId++) {
    ReleaseDC(hWndLetter[wmId], hdcLetter[wmId]);
}
```

Последняя надпись, которая нужна — надпись для нового слова. Ее создание ничем не примечательно:

```
// Рамка окна нового слова
W1 = CreateWindow("STATIC", 0,
    WS_CHILD | WS_VISIBLE | SS_ETCHEDFRAME,
    PX8, PY8, PW8, PH8, hWnd, 0, hAppInst, 0);
// Окно надписи нового слова
hWndWord = CreateWindow("STATIC", szPlayerName[wmId],
    WS_CHILD | WS_VISIBLE,
    4, 4, PW8 - 8, PH8 - 8, W1, 0, hAppInst, 0);
SendMessage(hWndWord, WM_SETFONT, (WPARAM)MainFont, 0);
SetWindowText(hWndWord, "Новое слово"); // Для отладки
```

Кнопки

На главной форме две кнопки — «Слово» и «Записать». Создание кнопки отличается от создания других окон только используемым предопределенным классом *BUTTON*. При создании кнопки ей сразу задается надпись, а после создания — новый шрифт.

Создаем кнопки при помощи функции *CreateWindow* и указываем их идентификаторы, определенные ранее в заголовочном модуле *KSGame.h*, в качестве идентификаторов меню:

```
// Кнопка "Слово"
W1 = CreateWindow("BUTTON", "&Слово",
    WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON | BS_NOTIFY,
    PX6, PY6, PW6, PH6,
    hWnd, (HMENU)IDB_WORD, hAppInst, 0);
SendMessage(hWndBWord, WM_SETFONT, (WPARAM)MainFont, 0);
// Кнопка "Записать"
W1 = CreateWindow("BUTTON", "&Записать",
    WS_CHILD | WS_VISIBLE | BS_PUSHBUTTON | BS_NOTIFY,
    PX7, PY7, PW7, PH7,
    hWnd, (HMENU)IDB_WRITE, hAppInst, 0);
SendMessage(hWndWrite, WM_SETFONT, (WPARAM)MainFont, 0);
```

Константа стиля кнопки *BS_PUSHBUTTON* предписывает создать нажимаемую кнопку (существуют еще кнопки-флажки и кнопки-переключатели), а константа *BS_NOTIFY* — кнопку, которая будет посылать сообщения о своих событиях родительскому окну.

Нас интересует сообщение *BN_CLICKED*, которое посылается при щелчке (нажатии и отпускании кнопки). Кроме него, кнопка посылает также и другие сообщения, например, получение и потерю фокуса. Сообщение передается в старшей части слова *wParam* и попадает в переменную *wmEvent*, а в младшей части этого слова передается идентификатор кнопки, который попадает в переменную *wmId*. В слово *lParam* записывается дескриптор окна кнопки.

В оконной процедуре главного окна обрабатываем сообщения кнопок:

```
    switch (wmId) {
    case IDB_WORD:
        if (wmEvent == BN_CLICKED) {
            DrawLetter(1, !LetterSel[1]); // Для отладки
            return 0;
        }
        break;
    case IDB_WRITE:
        if (wmEvent == BN_CLICKED) {
            DrawLetter(2, !LetterSel[2]); // Для отладки
            return 0;
        }
        break;
    case IDM_EXIT:
```

На этом построение интерфейса программы завершено. Отладочные операторы следует удалить или изменить.

Функциональность программы

Для реализации функциональности игры используем несколько вспомогательных классов. Класс добавляется в программу вручную или при помощи мастера. Чтобы добавить класс при помощи мастера, выбираем в меню *Insert—New Class…*, и вызываем диалог для добавления нового класса в программу. В поле *Name* вводим название нового класса и щелкаем кнопку *OK*. В программе появляется два файла — заголовочный и основной. В заголовочном модуле располагается объявление класса, а в основном — конструктор и деструктор.

Далее в класс по мере необходимости добавляются переменные и функции с использованием контекстного меню класса на вкладке *ClassView* или вручную.

Класс файла инициализации

Первый класс — это класс файла инициализации *Settings*, предназначенный для сохранения настроек (параметров) игры. Выбираем в меню *Insert—New Class…*, вводим название класса *Settings* и щелкаем кнопку *OK*.

Переходим на вкладку *ClassView*, вызываем контекстное меню класса, и выбираем *Add Member Function…*. В поле *Function Type* водим тип функции *void*, в поле *Function Declaration* вводим описание функции *PutPath(LPTSTR Path)*.

Открываем модуль описания класса *Settings.h*. Добавляем переменную, в которую будем записывать путь к файлу инициализации:

```
class Settings
{
private:
    TCHAR* pPath;
public:
    void PutPath(LPTSTR Path);
    Settings();
    virtual ~Settings();
};
```

Переходим в модуль *Settings.cpp*. Описываем конструктор и деструктор:

```
Settings::Settings()
{
    pPath = new TCHAR[1];
    pPath[0] = 0;
}

Settings::~Settings()
{
    delete[] pPath;
}
```

Описываем функцию *PutPath*:

```
void Settings::PutPath(LPTSTR Path)
{
    delete[] pPath;
    pPath = new TCHAR[1 + strlen(Path)];
    strcpy(pPath, Path);
}
```

Добавляем функцию для записи параметра. Тип функции *void*, описание функции *PutSetting(LPTSTR Value, LPTSTR Name)*. Описываем запись параметра:

```
void Settings::PutSetting(LPTSTR Value, LPTSTR Name)
{
    if (strlen(pPath) == 0) return;
    WritePrivateProfileString("KSGame", Name, Value, pPath);
}
```

Добавляем функцию для чтения параметра. Тип функции *void*, описание функции *GetSetting(LPTSTR Value, int Size, LPTSTR Name, LPSTR Default)*. Описываем чтение параметра:

```
void Settings::GetSetting(LPTSTR Value, int Size,
                          LPTSTR Name, LPSTR Default)
{
    Value[0] = 0;
    if (strlen(pPath) == 0) return;
    GetPrivateProfileString(KSGame", Name, Default, Value, Size, pPath);
}
```

Интегрируем класс в программу. Прежде всего включаем заголовочный модуль класса в модуль *KSGame.cpp*:

```
#include "Settings.h"
```

После описания прототипов, объявляем функцию для чтения параметров:

```
// Читает параметры из файла инициализации
void GetSettings()
{
}
```

Функция *GetSettings* получает путь к программе, формирует путь к каталогу программы, путь к файлу инициализации, путь к файлу базы данных, создает объект класса *Settings* и читает параметры. Для выполнения действий необходимы переменные, которые объявляем в *KSGame.h*:

```
// Путь к каталогу программы
TCHAR szAppPath[MAX_STRING];
// Путь к файлу инициализации
TCHAR szIniPath[MAX_STRING];
// Путь к базе данных
TCHAR szDBPath[MAX_STRING];
// Исходное слово
TCHAR szParentWord[] = "ГОРОД";
```

Возвращаемся в модуль *KSGame.cpp* и получаем путь к программе при помощи функции *GetWindowModuleFileName*:

```
void GetSettings()
{
    // Спецификация программы
    TCHAR Path[MAX_STRING];
    // Получаем спецификацию программы
    GetWindowModuleFileName(hWndMain, Path, MAX_STRING);
}
```

Для тестирования нужно вызвать функцию *GetSettings* в основной функции:

```
    if (!CreateMainForm(hInstance, nCmdShow)) {
        return FALSE;
    }
    GetSettings();
    while (GetMessage(&msg, NULL, 0, 0)) {
```

Возвращаемся в функцию и описываем извлечение пути к каталогу программы. Сначала нужно определить, где находится последний слэш. Он отделяет каталог программы от имени программы. Например, если путь к программе описывается строкой "*E:\PRO_VC\KSGame\Debug\KSGame.exe*", то последний слэш находится в позиции, которая завершает путь к каталогу. Найти последнее вхождение символа в строку можно при помощи функции *strrchr*:

```
    // Указатель на последний слэш
    LPTSTR p = strrchr(Path, 92);
    int m, i = 0, j = 0;
    if (p != 0) {
        // Позиция слэша
        m = p - Path;
    } else {
        m = strlen(Path);
    }
```

Далее нужно скопировать путь к программе до последнего слэша в переменную *szAppPath*, в которой каждый одиночный слэш должен быть заменен двумя. Например, если путь к программе равен "*E:\PRO_VC\KSGame\Debug*", то его нужно заменить на "*E:\\PRO_VC\\KSGame\\Debug*". Здесь это достигается посимвольным анализом строки *Path*:

```
    // Копируем путь к программе
    do {
        if (Path[j] != 34) {
            // Если не кавычка
            szAppPath[i++] = Path[j];
            // Один слэш заменяем двумя
            if (Path[j] == 92) szAppPath[i++] = Path[j];
        }
        j++;
    } while (j < m);
    // Добавляем ноль
    szAppPath[i] = 0;
```

Далее можно получить необходимые строки:

```
    // Копируем путь к программе
    strcpy(szIniPath, szAppPath);
    // Копируем путь к программе для базы данных
    strcpy(szDBPath, Path);
    // Добавляем имя файла инициализации
    strcat(szIniPath, "\\\\KSGame.ini");
    // Добавляем имя файла базы данных
    strcpy(szDBPath + m, "\\KSGame.mdb");
```

Дальше в этой процедуре создается объект класса *Settings*, устанавливается путь к файлу инициализации и читаются параметры:

```
    // Создаем файл инициализации
    Settings s;
    // Задаем путь к файлу инициализации
    s.PutPath(szIniPath);
    // Читаем параметры
    s.GetSetting(szParentWord, 6, "Word", "ГОРОД");
    s.GetSetting(szPlayerName[0], 80, "Name1", "Игрок 1");
    s.GetSetting(szPlayerName[1], 80, "Name2", "Игрок 2");
```

Для записи параметров ниже описываем новую функцию *PutSettings*. Она создает объект, устанавливает путь к файлу и записывает параметры:

```
// Записывает параметры в файл инициализации
void PutSettings()
{
    // Создаем файл инициализации
    Settings s;
    // Задаем путь к файлу инициализации
    s.PutPath(szIniPath);
    // Записываем параметры
    s.PutSetting(szParentWord, "Word");
    s.PutSetting(szPlayerName[0], "Name1");
    s.PutSetting(szPlayerName[1], "Name2");
}
```

Запись параметров происходит при завершении программы. Располагаем ее перед выходом из основной функции:

```
    PutSettings();
    return msg.wParam;
```

Следует убедиться, что файл инициализации создается и содержит следующие строки:

```
[KSGame]
Word=ГОРОД
Name1=Игрок 1
Name2=Игрок 2
```

Класс игрового поля

Добавляем новый класс *KSSpace*.

Класс игрового поля — это массив клеток, предназначенный для анализа допустимости положения новой буквы на поле. Описываем массив в заголовочном модуле класса *KSSpace.h*:

```
class KSSpace
{
private:
    int Cells[25];
public:
```

Добавляем в класс *KSSpace* элементы-функции:

Function Type	*Function Declaration*
int	get_Cell(int Index)
void	put_Cell(int Index, int NewValue)
int	IsValidCell(int Index)
void	Clear(LPTSTR Word)

Переходим в модуль *KSSpace.cpp* и описываем функцию *get_Cell*. Она возвращает клетку поля:

```
int KSSpace::get_Cell(int Index)
{
    if (Index < 0 || Index > 24) return 0;
    return Cells[Index];
}
```

Описываем функцию *put_Cell*. Она устанавливает клетку поля:

```
void KSSpace::put_Cell(int Index, int NewValue)
{
    if (Index < 0 || Index > 24) return;
    Cells[Index] = NewValue;
}
```

Функция *IsValidCell* проверяет допустимость клетки новой буквы. Прежде всего новая буква не может располагаться в области исходного слова. Клетка должна быть пустой и рядом с ней должна быть хотя бы одна заполненная клетка. Рассматриваем окрестность проверяемой клетки размером 3×3. Если проверяемая клетка находится на крае игрового поля, окрестность уменьшается до размера 2×3, 3×2 или 2×2.

```
int KSSpace::IsValidCell(int Index)
{
    // Проверяем клетки исходного слова
    if ((Index > 9) && (Index < 15)) return 0;
    // Клетка должна быть пустой
    if (Cells[Index] != 0) return 0;
    // Определяем строку проверяемой клетки
    int R = Index / 5;
    // Определяем столбец проверяемой клетки
    int C = Index % 5;
    // Проверяем окрестности клетки Index
    // Цикл по строкам окрестности
    for (int i = max(0, R - 1); i <= min(4, R + 1); i++) {
        // Цикл по столбцам окрестности
        for (int j = max(0, C - 1); j <= min(4, C + 1); j++) {
            // Индекс клетки окрестности
            int N = j + i * 5;
            if (Cells[N] != 0) {
                // Клетка рядом с занятой
                return 1;
            }
        }
    }
    return 0;
}
```

Последняя функция очищает игровое поле и записывает исходное слово:

```
void KSSpace::Clear(LPTSTR Word)
{
    int i, L = strlen(Word);
    for (i = 0; i < 25; i++) {
        Cells[i] = 0;
    }
    for (i = 0; i < L; i++) {
        Cells[i + 10] = Word[i];
    }
}
```

Класс пути нового слова

Добавляем новый класс *KSPath*.

Класс пути нового слова — это массив, описывающий клетки нового слова. Он предназначен для определения правильности пути нового слова по полю.

В заголовочном модуле класса *KSPath.h* описываем структуры данных:

```
class KSPath
{
private:
    // Путь нового слова по полю
    int Path[25];
    // Текущая длина пути
```

```
    int count;
    // Новое слово
    TCHAR Word[25];
public:
```

Добавляем в класс *KSPath* элементы-функции:

Function Type	*Function Declaration*
void	Add(int Index, int Letter)
int	AreNeighbours(int A, int B)
void	Clear()
int	IsValidLetter(int Index)
LPTSTR	get_Word()

Переходим в модуль *KSPath.cpp* и редактируем конструктор:

```
KSPath::KSPath()
{
    count = 0;
}
```

Описываем функцию добавления новой клетки слова:

```
// Добавляет в указанную клетку новую букву
void KSPath::Add(int Index, int Letter)
{
    Path[count] = Index;
    if (count == 0) {
        // Первая буква - прописная
        Word[count] = Letter & ~32;
    } else {
        // Следующие буквы - строчные
        Word[count] = Letter | 32;
    }
    count++;
    // Завершающий ноль
    Word[count] = 0;
}
```

Функция *AreNeighbours* проверяет соседство двух клеток. Для этого вычисляются расстояния между клетками *D1* и *D2*:

```
// Проверяет соседство двух клеток
int KSPath::AreNeighbours(int A, int B)
{
    int D1, D2;
    D1 = abs((A / 5) - (B / 5));
    D2 = abs((A % 5) - (B % 5));
    if ((D1 > 1) || (D2 > 1)) return 0;
    if ((D1 == 0) && (D2 == 0)) return 0;
    return 1;
}
```

Следующая функция очищает объект:

```
// Очищает объект
void KSPath::Clear()
{
    count = 0;
    Word[count] = 0;
}
```

Функция *IsValidLetter* проверяет допустимость очередной буквы слова. Буква должна быть новой и соседней с предыдущей. Другие правила для простоты не проверяются:

```
// Проверяет допустимость очередной буквы
int KSPath::IsValidLetter(int Index)
{
    int i;
    if (count > 0) {
        // Индекс не должен повторяться
        for (i = 0; i < count; i++) {
            if (Index == Path[i]) return 0;
        }
        // Индекс должен быть соседним с предыдущим
        if (!AreNeighbours(Path[count - 1], Index)) return 0;
    }
    return 1;
}
```

Последняя функция возвращает новое слово:

```
LPTSTR KSPath::get_Word()
{
    return Word;
}
```

Класс слов игры

Добавляем новый класс *KSWords*. Класс слов — это массив слов, добавляемых в процессе игры. Он проверяет новое слово на совпадение с имеющимися. В заголовочном модуле класса описываем структуры данных:

```
class KSWords
{
private:
    TCHAR Words[20][26];
    int count;
public:
```

Добавляем в класс *KSWords* элементы-функции:

Function Type	*Function Declaration*
int	get_Count()
void	Add(LPTSTR Word)
void	Clear()
int	IsValidWord(LPTSTR Word)

Переходим в модуль *KSWords.cpp* и описываем конструктор:

```
KSWords::KSWords()
{
    count = 0;
}
```

Функция, которая возвращает количество слов:

```
int KSWords::get_Count()
{
    return count;
}
```

Функция, которая добавляет новое слово:

```
void KSWords::Add(LPTSTR Word)
{
    strcpy(Words[count], Word);
    count++;
}
```

Функция, очищающая объект:

```
void KSWords::Clear()
{
    count = 0;
}
```

Последняя функция сравнивает слово с имеющимися:

```
// Сравнивает слово с имеющимися
int KSWords::IsValidWord(LPTSTR Word)
{
    int i;
    for (i = 0; i < count; i++) {
        if (strcmp(Word, Words[i]) == 0) {
            return 0;
        }
    }
    return 1;
}
```

Класс словаря

Добавляем новый класс *KSDictionary*.

Этот класс предназначен для поиска слов в базе данных. Саму базу данных предварительно нужно создать и разместить в папке приложения, так как приложение будет искать ее именно там.

База данных создается при помощи приложения Microsoft Access. Открыв его, указываем создание новой базы данных, далее указываем путь и название базы данных KSGame.mdb.

База данных состоит из одной таблицы Words. Таблица содержит два поля. Первое поле Word имеет тип «Текстовый» и является ключевым, размер поля 30. Второе поле Default имеет тип «Числовой», размер поля «Целое».

Открываем модуль *StdAfx.h* и добавляем описания, необходимые для использования интерфейса ADO (ActiveX Data Objects):

```
#include <windows.h>
#include <winable.h>
#import "c:\Program Files\Common Files\System\ADO\msado15.dll" \
   no_namespace rename("EOF", "EndOfFile")
```

Интерфейс ADO — это коллекция классов, предназначенных для доступа к базам данных. Мы используем объект *Recordset* для выполнения запросов.

Описываем структуры данных класса в модуле *KSDictionary.h*:

```
#include <string.h>
// Строка подключения
#define CONN_STRING "Provider=Microsoft.Jet.OLEDB.4.0;Data Source="

class KSDictionary
{
private:
    TCHAR szConnectionString[256];
    TCHAR szQuery[256];
public:
```

Добавляем в класс *KSDictionary* элементы-функции:

Function Type	*Function Declaration*
`int`	`IsValidWord(LPTSTR Word)`
`void`	`put_Path(LPTSTR Path)`
Следующие две функции имеют тип *private*	
`int`	`Find(LPTSTR Word)`
`int`	`Insert(LPTSTR Word)`

Переходим в модуль *KSDictionary.cpp* и описываем вспомогательную функцию для выполнения операций с COM-объектами (перед конструктором):

```
// Функция для проверки результата операции COM
inline void TESTHR(HRESULT _hr)
{
    if FAILED(_hr) _com_issue_error(_hr);
}
```

Описываем конструктор:

```
KSDictionary::KSDictionary()
{
    szConnectionString[0] = 0;
    szQuery[0] = 0;
}
```

Функция *IsValidWord* проверяет слово в словаре при помощи закрытой функции *Find*. Если слово не найдено, используется функция *MessageBox* для запроса пользователю на добавление нового слова с словарь. Если пользователь отвечает «Да», слово добавляется закрытой функцией *Insert*:

```
int KSDictionary::IsValidWord(LPTSTR Word)
{
    if (Find(Word)) {
        return 1;
    }
    int m = MessageBox(0,
        "Слово не найдено в словаре. Добавить его?",
        "KSGame Dictionary", MB_ICONQUESTION | MB_YESNO);
    if (m == IDNO) {
        return 0;
    } else {
        return Insert(Word);
    }
}
```

Функция *put_Path* создает строку подключения к базе данных:

```
void KSDictionary::put_Path(LPTSTR Path)
{
    strcpy(szConnectionString, CONN_STRING);
    strcat(szConnectionString, Path);
}
```

Функция *Find* создает строку запроса и далее пытается выполнить запрос при помощи *try*-блока. *Catch*-блок «ловит» любые ошибки, которые при этом могут возникнуть:

```
int KSDictionary::Find(LPTSTR Word)
{
    // Формируем строку запроса
    strcpy(szQuery, "SELECT * FROM Words WHERE Word='");
    strcat(szQuery, Word);
    strcat(szQuery, "'");
    // Пробуем выполнить запрос
    try {
        // Объект ADODB.Recordset
        _RecordsetPtr rs("ADODB.Recordset");
        // Открываем как команду
        TESTHR(rs->Open(szQuery, szConnectionString,
            adOpenStatic, adLockOptimistic, adCmdText));
        // Получаем число возвращенных записей
        int ret = rs->RecordCount;
        TESTHR(rs->Close());
        return (ret > 0);
    }
    catch (...) {
        MessageBox(0,
            "Ошибка при поиске в базе данных.",
            "KSGame Dictionary", MB_ICONSTOP);
    }
    return 0;
}
```

Последняя функция описывает процесс добавления слова в словарь:

```
int KSDictionary::Insert(LPTSTR Word)
{
    // Пробуем выполнить запрос
    try {
        // Объект ADODB.Recordset
        _RecordsetPtr rs("ADODB.Recordset");
        // Открываем как таблицу
        TESTHR(rs->Open("Words", szConnectionString,
            adOpenStatic, adLockOptimistic, adCmdTable));
        // Добавляем новую запись
        TESTHR(rs->AddNew());
        // Устанавливаем поле
        rs->Fields->GetItem("Word")->Value = Word;
        // Обновляем
        TESTHR(rs->Update());
        // Закрываем
        TESTHR(rs->Close());
        return 1;
    }
    catch (...) {
        MessageBox(0,
            "Ошибка при добавлении слова в базу данных.",
            "KSGame Dictionary", MB_ICONSTOP);
    }
    return 0;
}
```

Интегрируем классы в программу. Включаем заголовочные модули классов в модуль *KSGame.cpp* и объявляем переменные для объектов:

```
#include "KSSpace.h"
#include "KSPath.h"
#include "KSWords.h"
#include "KSDictionary.h"

// Игровое поле
KSSpace Space;
// Новое слово
KSPath Path;
// Слова
KSWords Words;
// Словарь
KSDictionary Dict;
```

В модуле *KSGame.h* описываем необходимые переменные игры, а также перечисление состояний:

```
// текущая клетка
int pCell = -1;
// текущая буква (русская)
int pLetter = 'А';
```

```
// текущий игрок
int pCurrentPlayer = 0;
// признак включения буквы
int pLetterIncluded = 0;
// Перечень состояний игры
enum GameState { stLetter, stWord, stGameOver };
// Состояние игры
GameState gmState = stLetter;
```

Описываем процесс начала новой игры. Для этого объявляем новую функцию в модуле *KSGame.cpp* после функции *PutSettings*:

```
// Начало игры
void GameNew(LPTSTR Word, LPTSTR Name1, LPTSTR Name2)
{
    pCell = -1;
    pLetter = 'А';  // Русская буква
    pCurrentPlayer = 0;
    // Игрок 0
    ClearList(0);
    PlayerCount[0] = 0;
    SetWindowText(hWndName[0], Name1);
    // Игрок 1
    ClearList(1);
    PlayerCount[1] = 0;
    SetWindowText(hWndName[1], Name2);
    strcpy(szParentWord, Word);
    int L = strlen(Word);
    for (int i = 0; i < L; i++) {
        // Выводим исходное слово
        CellText[i + 10] = Word[i];
        DrawCell(i + 10, 0);
    }
    Space.Clear(szParentWord);
    Path.Clear();
}
```

Вызываем функцию *GameNew* в основной функции после получения параметров из файла инициализации. Здесь же инициализируем библиотеку COM и устанавливаем путь к базе данных:

```
    ::CoInitialize(0);
    GetSettings();
    // Устанавливаем путь к базе данных
    Dict.put_Path(szDBPath);
    GameNew(szParentWord, szPlayerName[0], szPlayerName[1]);
```

Деинициализируем библиотеку COM перед завершением программы:

```
    PutSettings();
    ::CoUninitialize();
    return msg.wParam;
```

После функции *GameNew* описываем новую функцию *IsValidLetter*:

```
// Проверяет допустимость новой буквы
int IsValidLetter(int Index)
{
    if (!Path.IsValidLetter(Index)) return 0;
    // Клетка не должна быть пустой
    if (Space.get_Cell(Index) == 0) {
        // Если клетка пуста, то это только новая буква
        if (Index != pCell) return 0;
        pLetterIncluded = 1;
    }
    return 1;
}
```

Ниже описываем функцию, которая проверяет щелчок в клетку поля:

```
// Меняет букву в клетке
void CellClick(int Index)
{
    if (gmState == stLetter) {
        // Ввод новой буквы на поле
        if(Space.IsValidCell(Index)) {
            // Клетка допустима, очищаем предыдущую
            if (pCell >= 0) {
                CellText[pCell] = 32;
                DrawCell(pCell, 0);
            }
            // Новая текущая клетка
            pCell = Index;
            // Меняем букву в клетке
            CellText[pCell] = pLetter;
            // Рисуем клетку
            DrawCell(pCell, 1);
        }
    } else if (gmState == stWord) {
        // Ввод буквы нового слова
        if (IsValidLetter(Index)) {
            // Добавляем букву к слову
            Path.Add(Index, CellText[Index]);
            // Подсвечиваем букву на поле
            DrawCell(Index, 1);
            // Выводим слово
            SetWindowText(hWndWord, Path.get_Word());
        }
    }
}
```

Ниже описываем функцию очистки всех клеток поля:

```
void UnselectAll()
{
    for (int i = 0; i < 25; i++) DrawCell(i, 0);
}
```

Ниже описываем функцию для добавления нового слова:

```
// Добавляет новое слово
int AddWord(LPTSTR Word)
{
    // Признак добавления слова
    int ret = 0;
    // Слово в верхнем регистре
    char W[26];
    // Копируем слово
    strcpy(W, Word);
    // Переводим в верхний регистр
    int n = strlen(Word);
    for (int i = 0; i < n; i++) {
        W[i] = W[i] & ~32;
    }
    // Проверяем вхождение новой буквы в новое слово
    if (!pLetterIncluded) {
        MessageBox(hWndMain,
            "Новая буква должна входить в новое слово.",
            "KSGame", MB_ICONSTOP);
    } else {
        // Проверяем допустимость слова на поле
        if (!Words.IsValidWord(Word)) {
            MessageBox(hWndMain,
                "Слово уже добавлено.",
                "KSGame", MB_ICONSTOP);
        } else {
            // Проверяем слово в словаре
            if (!Dict.IsValidWord(W)) {
                MessageBox(hWndMain,
                    "Слово отсутствует в словаре.",
                    "KSGame", MB_ICONSTOP);
            } else {
                // Добавляем новое слово в список слов
                Words.Add(Word);
                // Записываем текущую букву на поле
                Space.put_Cell(pCell, pLetter);
                // Добавляем счет текущего игрока
                PlayerCount[pCurrentPlayer] += strlen(Word);
                // Выводим счет
                SetWindowText(hWndCount[pCurrentPlayer],
                    StrCount(pCurrentPlayer,
                    PlayerCount[pCurrentPlayer]));
                // Включаем новое слово в список
                AddToList(pCurrentPlayer, Word);
                // Сбрасываем признак добавления новой буквы
                pCell = -1;
                // Меняем текущего игрока
                pCurrentPlayer = (pCurrentPlayer + 1) % 2;
                // Успешное добавление
```

```
                ret = 1;
            }
        }
    }
    // Проверяем окончание игры
    if (Words.get_Count() == 20) {
        gmState = stGameOver;
        MessageBox(hWndMain,
            "Игра закончена.",
            "KSGame", MB_ICONINFORMATION);
    } else {
        gmState = stLetter;
    }
    return ret;
}
```

Далее в программе идет основная функция.

Корректируем функцию *CellProc*. Изменения касаются только события отпускания кнопки мыши:

```
            ReleaseCapture();
            // Меняем букву в клетке
            CellClick(index);
            return 0;
        case WM_PAINT:
```

Корректируем функцию *LetterProc*. Изменения касаются только события нажатия кнопки мыши:

```
    case WM_LBUTTONDBLCLK:
        // На клетку щелкнули
        if (LetterSel[index] == 0) {
            // Убираем существующее выделение
            for (i = 0; i < 32; i++) {
                if (LetterSel[i] != 0) {
                    DrawLetter(i, 0);
                }
            }
            DrawLetter(index, !LetterSel[index]);
            // Текущая буква
            pLetter = LetterText[index];
            // Если клетка задана, меняем ее букву
            if (pCell >=0) {
                CellClick(pCell);
            }
        } else {
            DrawLetter(index, !LetterSel[index]);
            // Захватываем события мыши
            SetCapture(hWnd);
        }
        return 0;
```

В функции *MainProc* обрабатываем события кнопок:

```
switch (message) {
case WM_COMMAND:
    wmId = LOWORD(wParam);
    wmEvent = HIWORD(wParam);
    switch (wmId) {
    case IDB_WORD:
        if (wmEvent == BN_CLICKED && gmState != stGameOver) {
            // Начало набора нового слова
            if (pCell >= 0) {
                // Гасим клетки
                UnselectAll();
                // Очищаем путь
                Path.Clear();
                // Новая буква не включена
                pLetterIncluded = 0;
                // Состояние - ввод слова
                gmState = stWord;
            } else {
                MessageBox(hWnd,
                    "Сначала нужно добавить новую букву.",
                    "KSGame", MB_ICONWARNING);
            }
            return 0;
        }
        break;
    case IDB_WRITE:
        if (wmEvent == BN_CLICKED && gmState == stWord) {
            // Конец набора нового слова
            if (AddWord(Path.get_Word())) {
                if (gmState == stGameOver) {
                    MessageBox(hWnd,
                        "Игра закончена.",
                        "KSGame", MB_ICONWARNING);
                }
            }
            // Очищаем слово
            SetWindowText(hWndWord, "");
            // Гасим клетки
            UnselectAll();
            return 0;
        }
        break;
    case IDM_EXIT:
```

На этом разработка основной части приложения закончена. Следующая часть — разработка диалогов.

Диалоги

Диалог — это модальное окно приложения, предназначенное для получения некоторых значений (параметров) от пользователя. Диалог приостанавливает выполнение основной программы до момента своего завершения.

Диалог — это ресурс. Открываем вкладку *ResourceView* и щелкаем ПРАВОЙ кнопкой мыши на *KSGame resources*. Выбираем *Insert…* и далее *Dialog*, после чего щелкаем кнопку *New*. Щелкаем ПРАВОЙ кнопкой мыши на название ресурса *IDD_DIALOG1* и выбираем *Properties*. В поле *ID* водим новое название ресурса *IDD_NEW*. Щелкаем ПРАВОЙ кнопкой мыши на окно диалога и выбираем *Properties*. В поле *Caption* вводим новый заголовок окна «Новая игра». Закрываем окно *Properties*.

Синяя пунктирная рамка — это область для размещения элементов диалога. Элементы управления автоматически притягиваются к ней, что обеспечивает необходимое выравнивание. Слева и сверху от окна диалога есть линейки, при помощи которых можно установить отступ синей рамки от края. Подводим указатель мыши к синей рамке, и перемещаем ее так, чтобы отступ сверху, слева и справа был равен 8 единицам, а снизу — 12-ти.

Зададим размер окна диалога, перемещая его границы и следя за показаниями в статусной строке. Ширина диалога равна 167, высота – 95. Заметим, что в диалоге используются необычные единицы измерения *DLU*, величина которых зависит от размера шрифта диалога.

Для размещения в диалоге элементов управления используется панель элементов управления под названием *Controls*, аналогичная панели инструментов Toolbox в Visual Basic. Она находится возле правого края окна среды разработки. Элемент нужно выбрать, а затем щелкнуть в окне диалога и «растягивать» элемент, удерживая кнопку мыши. При этом в статусной строке внизу справа отображается положение и размер элемента.

Размещаем в диалоге метку *Static Text*. Положение 8, 8, размер 92x10. Вызываем окно *Properties* для метки и вводим *Caption* — Исходное слово:.

Размещаем в диалоге поле *Edit Box*. Положение 8, 18, размер 92x12. Вызываем окно *Properties* для редактора, меняем *ID* на *IDC_WORD*.

Размещаем в диалоге метку *Static Text*. Положение 8, 34, размер 92x10. Вызываем окно *Properties* для метки и вводим *Caption* — Имя первого игрока:.

Размещаем в диалоге поле *Edit Box*. Положение 8, 44, размер 92x12. Вызываем окно *Properties* для редактора, меняем *ID* на *IDC_NAME1*.

Размещаем в диалоге метку *Static Text*. Положение 8, 60, размер 92x10. Вызываем окно *Properties* для метки и вводим *Caption* — Имя второго игрока:.

Размещаем в диалоге поле *Edit Box*. Положение 8, 71, размер 92x12. Вызываем окно *Properties* для редактора, меняем *ID* на *IDC_NAME2*.

Дважды щелкаем в окно диалога, чтобы вызвать окно *Properties* для окна. Устанавливаем начальное положение окна *X = 8*, *Y = 76*.

Выбираем в меню *Layout—Tab Order* или введем *Ctrl+D*. Поочередно щелкаем в первый сверху *Edit*, *Edit* под ним, нижний *Edit*, кнопку *OK* и кнопку *Cancel*. Цифры возле элементов показывают порядок переключения фокуса.

Чтобы посмотреть, как будет выглядеть окно на экране, нужно щелкнуть кнопку *Test* в нижнем левом углу окна среды разработки или ввести *Ctrl+T*:

Вызываем окно *Properties* для кнопки *Cancel* и в поле *Caption* меняем надпись кнопки на *Отмена*.

Сохраняем *Ctrl+S*.

Окно диалога готово. Закрываем среду разработки Visual C++ и открываем текстовый файл ресурсов *KSGame.rc* подходящим текстовым редактором.

Описание диалога должно иметь вид:

```
IDD_NEW DIALOG DISCARDABLE  8, 76, 167, 95
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Новая игра"
FONT 8, "MS Sans Serif"
BEGIN
    EDITTEXT        IDC_WORD,8,18,92,12,ES_AUTOHSCROLL
    EDITTEXT        IDC_NAME1,8,44,92,12,ES_AUTOHSCROLL
    EDITTEXT        IDC_NAME2,8,71,92,12,ES_AUTOHSCROLL
    DEFPUSHBUTTON   "OK",IDOK,115,8,50,14
    PUSHBUTTON      "Отмена",IDCANCEL,115,24,50,14
    LTEXT           "Исходное слово:",IDC_STATIC,8,8,92,10
    LTEXT           "Имя первого игрока:",IDC_STATIC,8,34,92,10
    LTEXT           "Имя второго игрока:",IDC_STATIC,8,60,92,10
END
```

Снова открываем среду разработки и проект *KSGame*.

Прежде чем вызывать диалог, нужно определить его диалоговую оконную процедуру. Описываем ее прототип в начале модуля *KSGame.cpp*:

```
// Прототипы функций модуля
int CreateMainForm(HINSTANCE hInstance, int nCmdShow);
LRESULT CALLBACK DialogNew(HWND, UINT, WPARAM, LPARAM);
```

В конце модуля описываем саму процедуру:

```
// Оконная процедура диалога "Новая игра"
LRESULT CALLBACK DialogNew(HWND hWnd, UINT message, WPARAM wParam,
                                                     LPARAM lParam)
{
    int wmId, wmEvent;
    switch (message) {
    case WM_INITDIALOG:
        return TRUE;
    case WM_COMMAND:
        wmId = LOWORD(wParam);
        wmEvent = HIWORD(wParam);
        switch (wmId) {
        case IDOK:
            EndDialog(hWnd, wmId);
            return TRUE;
        case IDCANCEL:
            EndDialog(hWnd, wmId);
            return TRUE;
        }
    }
    return FALSE;
}
```

Обратим внимание, что оконная функция диалогового окна возвращает либо *TRUE*, либо *FALSE*. Функция *EndDialog* завершает диалог. Второй параметр функции — код завершения диалога.

Диалог создается и вызывается при помощи функции *DialogBox* в оконной процедуре главного окна в ответ на команду *IDM_NEW*. Одновременно он связывается с оконной процедурой:

```
    case IDM_NEW:
        DialogBox(hAppInst,
            (LPCTSTR)IDD_NEW, hWnd, (DLGPROC)DialogNew);
        return 0;
    case IDM_EXIT:
```

Проект нужно построить *F7* и запустить *F5*. Далее в меню приложения выбираем «Игра—Новая».

Передача и возврат параметров диалога

В оконной процедуре диалога во время его создания передаем значения параметров элементам диалога при помощи функции *SetDlgItemText* и получаем обратно во время нажатия кнопки *OK* при помощи функции *GetDlgItemText*:

```
case WM_INITDIALOG:
    // передаем текущие параметры игры в диалог
    SetDlgItemText(hWnd, IDC_WORD, szParentWord);
    SetDlgItemText(hWnd, IDC_NAME1, szPlayerName[0]);
    SetDlgItemText(hWnd, IDC_NAME2, szPlayerName[1]);
    return TRUE;
case WM_COMMAND:
    wmId = LOWORD(wParam);
    wmEvent = HIWORD(wParam);
    switch (wmId) {
    case IDOK:
        // получаем новые параметры из диалога
        GetDlgItemText(hWnd, IDC_WORD, szParentWord, 6);
        GetDlgItemText(hWnd, IDC_NAME1, szPlayerName[0], 20);
        GetDlgItemText(hWnd, IDC_NAME2, szPlayerName[1], 20);
        GameNew(szParentWord, szPlayerName[0], szPlayerName[1]);
        EndDialog(hWnd, wmId);
        return TRUE;
```

Дополнительно нужно установить флажок *Uppercase* на вкладке *Styles* в окне свойств поля ввода *IDC_WORD* для того, чтобы пользователь мог вводить только прописные буквы.

Диалог «О программе...»

Дополнительно следует описать диалог «О программе...». Делаем это самостоятельно. В этом диалоге следует разместить значок приложения. Прежде всего в окно диалога добавляется элемент управления *Picture*. Далее в диалоге свойств этого элемента в списке *Type* выбираем *Icon*, а в списке *Image* выбираем большой значок приложения. Кроме значка, в диалоге следует разместить надпись «Игра "Королевский квадрат". Версия 1.0.». Из двух кнопок следует оставить только кнопку *OK*. Идентификатор диалога — *IDD_ABOUT*.

Оконная процедура должна иметь название *DialogAbout* и содержать обработку события *WM_INITDIALOG* и команды *IDOK*.

В меню приложения нужно добавить пункт меню «Справка» с подпунктом «О программе…». Идентификатор подпункта *IDM_ABOUT*.

Размеры и положение окна

Высота диалогового окна складывается из высоты заголовка, высоты меню, ширины границ и высоты собственно клиентской части. Первые три параметра являются системными, и они зависят от текущих настроек операционной системы. При создании главного окна приложения следует узнать эти параметры и вычислить правильную высоту окна. Ширина окна практически не зависит от системных настроек, поэтому ее можно вычислить на этапе конструирования окна.

Системные параметры возвращает функция *GetSystemMetrics*. Параметром функции является константа, указывающая на определяемую характеристику. Возвращается число типа *long*. Вычисляем сумму всех системных параметров и высоты клиентской части перед созданием главного окна в функции *CreateMainForm*:

```
// Высота окна
int H = GetSystemMetrics(SM_CYCAPTION);
H += GetSystemMetrics(SM_CYMENU);
H += GetSystemMetrics(SM_CYFRAME) + 326;
```

В последнем операторе 326 — это заранее подсчитанная высота клиентской части вместе с 3D рамкой.

Чтобы при старте программы главное окно приложения размещалось по центру экрана, нужно знать его ширину и высоту. Определяем их после вычисления высоты окна:

```
int SW = GetSystemMetrics(SM_CXSCREEN);
int SH = GetSystemMetrics(SM_CYSCREEN);
```

Теперь можно переопределить параметры окна при его создании:

```
hWndMain = CreateWindowEx(
    WS_EX_CLIENTEDGE, "KSMainWindow", szAppTitle,
    WS_POPUPWINDOW | WS_CAPTION,
     (SW - 515) / 2, (SH - H) / 2,
    515, H, 0, 0, hInstance, 0);
```

Акселераторы

Акселераторы — это клавиши и сочетания клавиш для быстрого выполнения пунктов меню. Акселераторы являются элементами таблицы акселераторов ресурсов приложения.

Откроем вкладку *ResourceView* и щелкнем ПРАВОЙ на *KSGame resources*. В контекстном меню выберем *Insert…*, далее *Accelerator* и щелкнем кнопку *New*.

Дважды щелкнем на пустую полоску таблицы акселераторов. Появится диалог *Properties*. Щелкнем кнопку *Next Key Typed* и введем сочетание *Ctrl+N*. В поле *ID* впишем идентификатор *IDM_NEW*. Закроем диалог *Properties*. В результате в таблице акселераторов должна появиться запись

IDM_NEW   Ctrl+N   VIRTKEY

Обратим внимание на то, что идентификатор акселератора должен в точности соответствовать идентификатору соответствующего пункта меню. Идентификатор таблицы акселераторов должен быть *IDR_ACCELERATOR1*.

Самостоятельно добавляем акселератор для выхода из программы — клавишу *Escape*, идентификатор *IDM_EXIT*.

Переходим в основную функцию и объявляем таблицу акселераторов:

```
MSG msg;
HACCEL hAccelTable;
```

Таблицу акселераторов необходимо загрузить перед циклом обработки сообщений, а сам цикл изменить, добавив в него проверку акселераторов:

```
hAccelTable = LoadAccelerators(hInstance, (LPCTSTR)IDR_ACCELERATOR1);
while (GetMessage(&msg, NULL, 0, 0)) {
    if (!TranslateAccelerator(hWndMain, hAccelTable, &msg)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

На этом построение приложения можно считать завершенным.

Заключение

Рассмотренный пример построения приложения с нуля является скорее исключением, чем правилом использования системы программирования Microsoft Visual C++. Начальная часть работы по созданию приложения обычно выполняется тем или иным мастером среды. Например, если при создании приложения мастером Win32 Application на шаге 1 выбрать *A typical "Hello World!" application*, будет построен шаблон приложения типа *SDI*, имеющий окно, меню, значки и диалог «О программе…».

Целью настоящего пособия является прояснение основных принципов создания и функционирования программ в среде Windows. Обычно же для создания программы используется мастер MFC, с помощью которого создается приложение на основе библиотеки классов MFC. Классы этой библиотеки в значительной мере упрощают решение большинства задач, возникающих в ходе разработки приложений. О том, как построить приложение с помощью библиотеки MFC, рассказывается в учебном пособии «Microsoft Visual C++. Приложение MFC с нуля».

Приложения

Стили окна Windows

Таблица 1. Основной стиль окна

Стиль	*Описание*
WS_BORDER	Окно с тонкой границей (рамкой).
WS_CAPTION	Окно с заголовком (включает WS_BORDER).
WS_CHILD WS_CHILDWINDOW	Дочернее окно. Не имеет меню. Не сочетается с WS_POPUP.
WS_CLIPCHILDREN	Исключает область дочерних окон во время рисования главного окна.
WS_CLIPSIBLINGS	Исключает наложенные дочерние окна из процедуры рисования данного дочернего окна.
WS_DISABLED	Недоступное для пользователя окно.
WS_DLGFRAME	Окно с границей, присущей диалоговым окнам. Окно не имеет заголовка.
WS_GROUP	Определяет первый элемент группы элементов управления. Все последующие элементы включаются в группу до тех пор, пока не встретится окно с WS_GROUP.
WS_HSCROLL	Окно с горизонтальной линейкой прокрутки.
WS_ICONIC WS_MINIMIZE	Создает минимизированное окно, то есть окно, свернутое в значок.
WS_MAXIMIZE	Создает максимизированное окно.
WS_MAXIMIZEBOX	Окно имеет кнопку «Развернуть». Не сочетается с WS_EX_CONTEXTHELP. Требует WS_SYSMENU.
WS_MINIMIZEBOX	Окно имеет кнопку «Свернуть». Не сочетается с WS_EX_CONTEXTHELP. Требует WS_SYSMENU.
WS_OVERLAPPED WS_TILED	Перекрывающееся окно. Имеет заголовок и границу.
WS_OVERLAPPEDWINDOW WS_TILEDWINDOW	Перекрывающееся окно с WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX и WS_MAXIMIZEBOX.
WS_POPUP	Всплывающее окно. Не сочетается с WS_CHILD.
WS_POPUPWINDOW	Всплывающее окно с WS_BORDER, WS_POPUP и WS_SYSMENU. Сочетание WS_POPUPWINDOW и WS_CAPTION создают окно с заголовком.
WS_SIZEBOX WS_THICKFRAME	Окно имеет границу для изменения размера.
WS_SYSMENU	Окно имеет системное меню. Требует WS_CAPTION.
WS_TABSTOP	Элемент управления, который получает фокус при помощи клавиши TAB.
WS_VISIBLE	Окно изначально видимо на экране.
WS_VSCROLL	Окно имеет вертикальную линейку прокрутки.

Таблица 2. Расширенный стиль окна (выборочно)

Стиль	*Описание*
WS_EX_ACCEPTFILES	Принимает файлы Drag and Drop.
WS_EX_APPWINDOW	Окно отображается в панели задач.
WS_EX_CLIENTEDGE	Границы окна образуют углубление, а не выступ.
WS_EX_CONTEXTHELP	В заголовке появляется кнопка «?». Когда пользователь щелкает её, к курсору добавляется знак вопроса. Объекту, на который пользователь щелкнет таким курсором, посылается сообщение WM_HELP. Не сочетается с WS_MAXIMIZEBOX и WS_MINIMIZEBOX.
WS_EX_CONTROLPARENT	Окно содержит дочерние окна, которые должны получать фокус при нажатии TAB.
WS_EX_DLGMODALFRAME	Окно имеет двойную рамку.
WS_EX_LAYERED	Windows 2000/XP: Слоёное главное окно. Несовместимо с CS_OWNDC и CS_CLASSDC.
WS_EX_MDICHILD	Создает дочернее окно MDI.
WS_EX_NOACTIVATE	Windows 2000/XP: Окно всегда находится на заднем плане. Не появляется в панели задач.
WS_EX_NOINHERITLAYOUT	Windows 2000/XP: Окно не передает свое расположение дочерним окнам.
WS_EX_NOPARENTNOTIFY	Дочернее окно не посылает WM_PARENTNOTIFY родительскому окну при создании и уничтожении.
WS_EX_OVERLAPPEDWINDOW	= WS_EX_CLIENTEDGE + WS_EX_WINDOWEDGE.
WS_EX_PALETTEWINDOW	= WS_EX_WINDOWEDGE + WS_EX_TOOLWINDOW + WS_EX_TOPMOST.
WS_EX_RIGHTSCROLLBAR	Вертикальная линейка прокрутки находится справа.
WS_EX_STATICEDGE	Окно с трехмерной границей, не принимающее фокус.
WS_EX_TOOLWINDOW	Окно панели инструментов.
WS_EX_TOPMOST	Окно, располагаемое на самом верхнем уровне в пределах приложения.
WS_EX_TRANSPARENT	Прозрачное окно (показывает дочерние окна).
WS_EX_WINDOWEDGE	Границы окна образуют выступ.

Классы элементов управления

Таблица 3. Предопределенные классы окон

Класс окна	*Описание*
BUTTON	Создает кнопку.
COMBOBOX	Создает выпадающий список.
EDIT	Создает окно редактирования.
LISTBOX	Создает список.
MDIClient	Создает дочернее окно MDI.
RichEdit	Создает редактор текста rtf версии 1.
RICHEDIT_CLASS	Создает редактор текста rtf версии 2 или 3.
SCROLLBAR	Создает линейку прокрутки.
STATIC	Создает текст, картинку или рамку для других элементов.

Константы сообщений

Таблица 4. Константы нажатых клавиш и кнопок мыши

Константа	***Описание***
MK_CONTROL	Нажата клавиша Ctrl.
MK_LBUTTON	Нажата левая кнопка мыши.
MK_MBUTTON	Нажата средняя кнопка мыши.
MK_RBUTTON	Нажата правая кнопка мыши.
MK_SHIFT	Нажата клавиша Shift.