Министерство образования и науки РФ

Федеральное государственное бюджетное образовательное учреждение
высшего образования «Томский государственный университет систем управления
и радиоэлектроники» (ТУСУР)

Специализированная подготовка разработчиков бизнес приложений

Методические указания по выполнению
лабораторных работ и заданий самостоятельной подготовки
для студентов направлений
09.03.01 - Информатика и вычислительная техника
09.03.02 - Информационные системы и технологии

Томск
2018

РАССМОТРЕНО И УТВЕРЖДЕНО на заседании кафедры экономической математики, информатики и статистики факультета вычислительных систем ТУСУР.
Протокол № 7 от «26» апреля 2018 г.

В учебном курсе рассматриваются вопросы подготовки разработчиков бизнес приложений для студентов выпускного курса; проводится знакомство с предметными областями их будущей профессиональной деятельности. В качестве предметной области выбрана автоматизация процесса торговли, при этом рассматривается техническое и программное обеспечение не только непосредственно торгово-закупочной деятельности, но и техническое и программное обеспечение банковских операций, электронных платежей и программная поддержка торговли через интернет. Также в рассмотрение включены проблемно-ориентированные вычислительные системы; основные принципы разработки программных систем; обучение основам создания законченных программных продуктов и программных комплексов.
Предназначено для студентов, магистрантов и аспирантов информационных направлений технических вузов.

Составитель:
зав.кафедрой ЭМИС Боровской И.Г.

СОДЕРЖАНИЕ

1. ДИСЦИПЛИНА «СПЕЦИАЛИЗИРОВАННАЯ ПОДГОТОВКА РАЗРАБОТЧИКОВ БИЗНЕС ПРИЛОЖЕНИЙ» 4
1.1. Цели преподавания дисциплины 4
1.2. Задачи изучения дисциплины 4
1.3. Требования к результатам освоения дисциплины 4
2. СОДЕРЖАНИЕ И РЕКОМЕНДАЦИИ ПО ВЫПОЛНЕНИЮ ЛАБОРАТОРНЫХ РАБОТ 5
2.1. Одномерный штрих-код 5
2.2. Двумерный штрих-код 9
2.3. Сканеры и принтеры штрих-кодов 15
2.4. Магнитные карты 16
2.5. RFID системы 18
2.6. Бесконтактные смарт-карты 22
2.7. Фискальные регистраторы и POS системы 22
2.8. Банкоматы и платежные терминалы 25
2.9. Видеонаблюдение 27
2.10. Антикражные ворота 31
3. ЗАДАНИЯ, ВЫНОСИМЫЕ НА САМОСТОЯТЕЛЬНУЮ РАБОТУ, И РЕКОМЕНДАЦИИ ПО ВЫПОЛНЕНИЮ 35
СПИСОК ЛИТЕРАТУРЫ 52

1. Дисциплина «Специализированная подготовка разработчиков бизнес приложений»

1.1. Цели преподавания дисциплины

Целью учебного курса «Специализированная подготовка разработчиков бизнес приложений» является ознакомление студентов выпускного курса с предметными областями их будущей профессиональной деятельности. В качестве предметной области выбрана автоматизация процесса торговли, при этом рассматривается техническое и программное обеспечение не только непосредственно торгово-закупочной деятельности, но и техническое и программное обеспечение банковских операций, электронных платежей и программная поддержка торговли через интернет. Также в рассмотрение включены проблемно-ориентированные вычислительные системы; основные принципы разработки программных систем; обучение основам создания законченных программных продуктов и программных комплексов; изучение методов создания приложений для операционных систем семейства Windows с использованием средств автоматизированного программирования.

1.2. Задачи изучения дисциплины

Основная задача изучения данного курса состоит в том, чтобы дать студентам представление о предметной области их профессиональной деятельности. Кроме того, одной из важных задач данного курса, является развитие творческой самостоятельности студентов. Лекционный материал предназначен для объяснения ключевых и наиболее сложных моментов разработки бизнес-приложений и предполагает большую самостоятельную работу с литературой. Лабораторные работы должны помочь студенту получить практические навыки разработки программных систем на примере объектно-ориентированной операционной системы Windows, с использованием как Win32 API, так и MFC в частности.

1.3. Требования к результатам освоения дисциплины

В результате изучения дисциплины студент должен:

знать: методологию построения бизнес-приложений с привлечением алгоритмов различной сложности; современные парадигмы программирования; конструктивные компоненты и структуру компьютерных программ.

уметь: использовать приемы и методы разработки программного обеспечения на основе современных технологий программирования.

владеть: навыками применения алгоритмических языков высокого уровня при решении широкого круга практических задач.

2. Содержание и рекомендации по выполнению лабораторных работ

2.1. Одномерный штрих-код

Задание.

Создать приложение, которое использует следующие ресурсы: строковый ресурс, пиктограмма, курсор мыши, графическое изображение типа bitmap. Строковый ресурс используется в заголовке окна приложения, пиктограмма выводится при минимизации окна, курсор мыши меняет свой вид при щелчке левой клавишей мыши, а изображение bitmap используется для фона окна, который меняется при щелчке правой клавишей мыши.

Пояснения и указания.

Подавляющее большинство Windows программ включает множество графических элементов, именуемых ресурсами (resources) Windows. Перечислим виды ресурсов в Win32 API:

1) меню, диалоги, панели управления, акселераторы;
2) иконки, битовые изображения, курсоры;
3) строковые таблицы, таблица версии приложения;
4) ресурсы, определяемые разработчиком.

Все перечисленные ресурсы размещаются в *файле ресурсов*, что представляет собой одну из самых характерных особенностей Windows программирования. Именно с помощью файлов ресурсов большинство приложений определяет визуальные элементы своего пользовательского интерфейса – меню, диалоговые окна, надписи, растровые изображения и прочее.

Файлы ресурсов, имеющие расширение **.rc* создаются по известным спецификациям, в текстовом формате. Это позволяет создавать файл ресурсов приложения либо вручную – любым текстовым редактором, либо специальным редактором ресурсов, что предпочтительнее. Затем этот файл компилируется *компилятором ресурсов*. Полученный в результате файл с расширением **.res* далее компонуется с остальными частями приложения – **.obj* и **.lib* файлами, образуя единый двоичный образ, содержащий выполняемый программный код и информацию о ресурсах. При этом в заголовке каждого Windows приложения формируется специальная *таблица ресурсов*. Эта таблица используется Windows для поиска и загрузки ресурсов в оперативную память.

Хотя ресурсы являются данными и хранятся в *exe* файле программы, но расположены они не в сегменте данных (DS), где хранятся обычные данные исполняемых модулей. Таким образом, к ресурсам нет непосредственного доступа через переменные, определенные в исходном тексте программы. Они должны быть явно загружены из *exe* файла в память.

Основная причина такого построения – экономия оперативной памяти. Действительно, трудно придумать ситуацию, когда *абсолютно все* ресурсы могут одномоментно понадобиться приложению. А раз в каждый момент нужны только некоторые, то более разумно – подгружать их по мере надобности.

Нужно сказать, что ресурсы не обязательно должны компоноваться с исполняемым файлом приложения, они также могут сводиться в отдельную библиотеку DLL. Преимущество этого подхода заключается в том, что при изменении файла ресурсов не требуется перекомпилировать все приложение, а нужно только заменить DLL файл. Можно также сопровождать приложение несколькими DLL, представляющими ресурсы на различных языках. Именно так поступают разработчики при создании приложения, предназначенного для работы в многоязыковой среде.

Еще одно преимущество – с точки зрения технологии программирования – использования ресурсов состоит в том, что локализация приложения в этом случае требует наименьших затрат.

Меню приложения. Меню – это наиболее консервативная и, одновременно, наиболее важная часть пользовательского интерфейса. Трудно найти Windows приложение, у которого отсутствует меню, – это считается “дурным тоном” у разработчиков.

Виды меню. Обычно выделяют четыре вида меню:

1. Главное меню окна или меню верхнего уровня (*top-level menu*). Это меню представляет собой горизонтальную строку, которая расположена непосредственно под заголовком окна и состоит из нескольких элементов.

2. Подменю (*submenu*) появляется под главным меню при выборе одного из его элементов.

3. Плавающее меню (*floating menu*). Оно не связано с главным меню, и может быть создано в любом месте экрана как независимое всплывающее меню.

4. У подавляющего большинства главных окон приложений в левой части заголовка находится пиктограмма. Щелчок мышью по ней приводит к появлению системного меню (*system menu*), которое похоже на подменю главного меню приложения. Обычно системное меню всех приложений одинаково, с его помощью можно минимизировать или максимизировать окно приложения, перемещать его и прочее. Однако приложение имеет возможность изменить системное меню, дополняя его новыми строками, или удаляя существующие.

Возможные состояния пунктов меню

1. При выборе пункта меню, строка инвертирует цвет. Это применимо к пунктам меню всех видов.

2. Пункты всплывающих меню могут находиться в состоянии *отмечено* (checked), при этом слева от текста пункта меню выставляется специальная метка. Эта метка позволяет пользователю узнать о том, какие опции программы выбраны из этого меню. Пункты главного меню *не могут быть отмечены*.

3. Пункты меню могут находиться в состоянии *активно* (enabled), *неактивно* (disabled) и *недоступно* (grayed). Имейте в виду, что пункты, помеченные как активные или неактивные, выглядят для пользователя одинаково, а недоступный пункт меню выводится в сером цвете.

Только при выборе активных пунктов меню Windows генерирует сообщения, поступающие в функцию окна, содержащего это меню.

Итак, каждый пункт меню определяется тремя характеристиками:

1. Внешний вид пункта меню, другими словами – то, что будет отображено в меню. Это может быть либо строка текста, либо графический, точнее битовый образ.

2. Атрибут пункта меню, который определяет, является ли данный пункт активным, недоступным или отмеченным.

3. Уникальный числовой идентификатор, который Windows посылает в функцию окна, когда пользователь выбирает данный пункт меню.

Сообщения от пунктов меню

При выборе пунктов меню Windows генерирует сообщение **`WM_COMMAND`**. Это сообщение можно назвать сообщением пользовательского интерфейса, поскольку оно посылается всякий раз, когда пользователь что-то выбирает, что-то меняет и так далее. В общем случае это сообщение имеет параметры:

```
WM_COMMAND
wEvent  = HIWORD(wParam);
wID     = LOWORD(wParam);
hWndCtl = (HWND)lParam;
```

Параметр **wEvent** именуется нотификационным кодом или кодом извещения. Для пунктов меню он *всегда* равен нулю. Параметр **wID** – это уникальное числовое значение, ассоциированное с каждым элементом. В случае меню, он содержит идентификатор пункта меню. Параметр **hwndCtl** представляет собой дескриптор окна элемента управления. Для пунктов меню он *всегда* равен NULL.

При выборе пунктов системного меню генерируется сообщение **WM_SYSCOMMAND**, которое имеет такие же параметры.

Напомним еще раз, что сообщение **WM_COMMAND** поступает в функцию окна, когда пункт меню *уже выбран*. Однако решаемая вами задача может потребовать знания, какой пункт меню *выбирается* пользователем в настоящий момент. Допустим, вы хотите динамически менять внешний вид пункта меню, когда пользователь наводит на него курсор мыши. В этом случае функция окна должна обрабатывать сообщение **WM_MENUSELECT**, которое поступает до **WM_COMMAND** и имеет параметры:

```
Item   = LOWORD(wParam);
fFlags = HIWORD(wParam);
hMenu  = (HMENU)lParam.
```

Параметры **Item** и **hMenu** соответствуют идентификатору и дескриптору выбираемого пункта меню, а значение **fFlags** содержит флаги состояния этого пункта.

Создание главного меню приложения

Добавление меню к программе – сравнительно простая задача Windows программирования. Структура меню достаточно просто определяется в описании ресурсов. Каждому пункту меню присваивается числовой идентификатор. Обеспечение уникальности идентификаторов берут на себя современные редакторы ресурсов, вам остается придумать мнемонические имена этим идентификаторам, например **ID_EXIT.**

Для создания меню применяются три метода:

1. Шаблон меню создается в файле ресурсов приложения, а затем загружается при создании главного окна приложения. Этот способ подходит для статических меню, неменяющихся или незначительно меняющихся в процессе работы программы.

2. Меню без шаблона создается динамически только при помощи функций Win32 API. Этот способ подходит для приложений, существенно меняющих внешний вид меню, когда разработать подходящий шаблон заранее не представляется возможным.

3. Шаблон меню подготавливается непосредственно в оперативной памяти и с помощью специальных функций подключается к приложению.

Мы, преимущественно, будем использовать первый способ. Как было сказано, прежде чем подключить меню к приложению, его нужно загрузить из ресурсов. Последовательность действий может быть такой:

```
HMENU hMenu = LoadMenu(g_hInst, "Main_Menu");
HWND   hWnd = CreateWindowEx(
       WS_EX_OVERLAPPEDWINDOW,
       szClass, szTitle, WS_OVERLAPPEDWINDOW,
       0, 0, 100, 100,
       NULL, hMenu, g_hInst, NULL);
if (!hWnd) return -1;
```

Заметим, что при указании ресурсов приложения чаще используются числовые значения, идентифицирующие ресурсы, а не строковые описания, например, **IDR_MAIN_MENU**. В этом случае следует использовать макрос **MAKEINTRESOURCE()** для преобразования идентификатора ресурса в строковое описание. Пример загрузки меню выглядит так:

```
HMENU hMenu = LoadMenu(g_hInst,
              MAKEINTRESOURCE(IDR_MAIN_MENU));
```

Обработчик сообщений меню несложен и может выглядеть следующим образом:

```
case WM_COMMAND: {
    int wID = LOWORD(wParam);
    switch (wID) {
       case ID_EXIT :
           DestroyWindow(hWnd);
           break;
    }
    return 0;  }
```

Функции для работы с меню

1. Если окно **hWnd** приложения имеет главное меню, то получить его дескриптор можно с помощью функции

```
HMENU hMenuMain = GetMenu(hWnd);
```

Затем можно получить дескрипторы всех подменю, вызывая функцию

```
HMENU hSubMenu = GetSubMenu(hMenuMain, nPos);
```

Здесь целочисленное значение **nPos** изменяется от 0 до **nMenu**-1, где переменная **nMenu** обозначает количество подменю. Узнать это значение можно с помощью универсальной функции

```
nMenu = GetMenuItemCount(HMENU hMenu),
```

которая возвращает количество элементов в *любом* меню, а не только в главном.

2. Для полной замены меню у главного окна приложения нужно воспользоваться функцией

```
BOOL SetMenu (HWND hWnd, HMENU hMenuNew),
```

которая позволяет прикрепить к окну **hWnd** новое меню с дескриптором **hMenuNew.**

В этом случае "старое" меню требуется разрушить вызовом функции **DestroyMenu(hMenu)**. Если же меню прикреплено к окну при создании последнего (см. пример предыдущего пункта), Windows автоматически разрушит такое меню при закрытии окна.

3. Изменить состояние пункта меню с идентификатором **uID** можно с помощью вызова функции

```
EnableMenuItem(hMenu, uID, uEnable);
```

Параметр **uEnable** может принимать одно из следующих значений:

MF_DISABLED (недоступно), **MF_ENABLED** (активно) или **MF_GRAYED** (недоступно и отмечено серым цветом). Это значение должно быть объединено с константой **MF_BYCOMMAND**, если параметр **uID** задает идентификатор меню, либо с константой **MF_BYPOSITION**, если **uID** задает номер позиции в меню.

4. Если вы изменяете состояние какого-либо пункта *главного меню* окна **hWnd**, то следует выполнить перерисовку меню вызовом функции

```
DrawMenuBar(hWnd);
```

Перерисовывать подменю не требуется, поскольку это делается Windows автоматически при каждом вызове подменю.

5. Программный интерфейс Win32 содержит две универсальные функции

```
BOOL GetMenuItemInfo(HMENU hMenu, UINT uItem,
       BOOL fByPosition, MENUITEMINFO* pMI)
BOOL SetMenuItemInfo(HMENU hMenu, UINT uItem,
       BOOL fByPosition, MENUITEMINFO* pMI)
```

первая из которых возвращает полную информацию о пункте любого меню, а вторая – позволяет совершить все необходимые операции с этим пунктом. Параметр **uItem** является позицией в меню, если переменная **fByPosition** имеет ненулевое значение, либо идентификатором – в противном случае. Исчерпывающее описание полей структуры **MENUITEMINFO** вы можете найти в любой справочной системе по Win32 API.

2.2. Двумерный штрих-код

Задание. Создать приложение, позволяющее выводить текст в рабочую область окна, текст выдавать различным цветом. Необходимо отследить появление в очереди следующих сообщений: WM_CREATE, WM_DESTROY, WM_PAINT, WM_MOVE, WM_SIZE.

Пояснения и указания.

Создание главного окна. Создание главного окна приложения производится вызовом функции **`CreateWindow()`** или **`CreateWindowEx()`**.

Для многих функций Win32 API содержит два варианта, при этом один из них имеет суффикс **`Ex`**. Именно последние являются истинными Win32 функциями, их использование предпочтительнее. Функции без суффикса, как правило, унаследованы из Win16 и не являются 32-х разрядными. Кроме того, фирма Microsoft предупреждает, что она может прекратить поддерживать такие функции в будущем.

Кратко укажем значения параметров функции **`CreateWindowEx()`** (полную информацию о параметрах можно найти, например, в справочной системе компилятора VC++):

`DWORD dwExStyle`	дополнительные стили окна, определяющие, главным образом, вид обрамления окна;
`LPCTSTR lpClassName`	имя зарегистрированного ранее класса окна;
`LPCTSTR lpWindowName`	текст заголовка окна;
`DWORD dwStyle`	основные стили окна, определяющие его внешний вид.
`int x, y, nWidth, nHeight`	координаты левого-верхнего угла окна, его ширина и высота. Задаются в пикселях;
`HWND hWndParent`	дескриптор родительского окна или NULL для окон верхнего уровня (главных окон);
`HMENU hMenu`	дескриптор меню приложения. Если меню отсутствует, то NULL;
`HINSTANCE hInstance`	дескриптор экземпляра приложения, полученного через параметр WinMain();
`LPVOID lpParam`	указатель на дополнительную информацию, ассоциированную с данным окном. Не используется, как правило, поэтому NULL.

В Windows поддерживается строгая иерархия окон, при этом *каждое* окно должно чему-либо принадлежать, т.е. иметь родительское окно. Единственным исключением является специальное окно под названием “DeskTop”, которое создается операционной системой при ее старте. На этом окне располагаются все видимые объекты самой Windows. Когда вы указываете нулевое значение для **`hWndParent`**, это означает, что истинным родительским окном будет “DeskTop”.

В Windows существуют *три основных стиля* окон:

- главные окна приложений имеют стиль **`WS_OVERLAPPEDWINDOW`**. Как правило, они располагаются на “DeskTop” и именуются окнами верхнего уровня;

- временные окна имеют стиль **`WS_POPUPWINDOW`**. Они принадлежат главным окнам приложений и служат обычно для вывода какой-либо информации. Типичный пример – диалоговые панели;

- дочерние окна, имеющие стиль **`WS_CHILD`**, не могут быть перемещены за границы родительского окна. Типичный пример – элементы управления диалогов.

Вернемся к точке (6) на нашей схеме. Если создание окна выполнено успешно, функция **CreateWindowEx()** возвращает его дескриптор. Под дескриптором окна следует понимать уникальное, в рамках операционной системы, значение, которое ассоциировано с каждым окном. Именно через этот дескриптор производятся все операции с конкретным окном.

В этой точке Windows, в своем ядре, создает *персональную очередь сообщений* для только что созданного окна. Сюда операционная система будет помещать сообщения, адресованные данному окну.

Для того, чтобы окно появилось на экране нужно вызвать функцию **ShowWindow(hWnd, nCmdShow)**, которая принимает значение дескриптора окна и параметр, переданный через заголовок **WinMain()**. После этого, мы заставляем окно перерисовать свое содержимое, обращаясь к функции **UpdateWindow(hWnd)**, которая помещает сообщение **WM_PAINT** в очередь сообщений окна **hWnd.**

Цикл обработки сообщений.

Ключевая точка каждого Windows приложения – цикл обработки сообщений. Как он работает?

Функция **GetMessage()** постоянно просматривает очередь сообщений и, если она пуста, не делает возврата, а передает управление Windows. Приложение находится в состоянии простоя или, как говорят, в состоянии "**idle**". Но как только в очереди появляется сообщение, **GetMessage()** заполняет соответствующие поля структуры **MSG**, удаляет это сообщение из очереди и возвращает *ненулевое* значение. В этом случае говорят – **GetMessage()** выбирает сообщение из очереди. После этого, управление передается функции **DispatchMessage(&msg)**, которая пересылает указатель на заполненную ранее структуру **MSG** обратно ядру Windows. Операционная система блокирует **DispatchMessage()** и вызывает оконную функцию (указатель на нее известен Windows) с параметрами, соответствующими полям структуры **MSG**. Функция окна выполняет действия, которые вы указали (или не указали) для обработки данного сообщения. После этого функция окна возвращает управление Windows, которая, в свою очередь, делает возврат из **DispatchMessage()**. Приложение возвращается к началу цикла **while**, т.е. вновь входит в функцию **GetMessage()**, ожидая следующего сообщения. Так продолжается до тех пор, пока в очереди сообщений не появится **WM_QUIT**. В этом случае **GetMessage()** возвращает *нулевое* значение, цикл **while** заканчивается, и приложение завершается.

Цикл обработки сообщений и функция окна работают последовательно. Пока функция окна проводит обработку текущего сообщения, функция **GetMessage()** не может выбрать другое сообщение, вследствие блокировки цикла в **DispatchMessage()**. Значит, прежде чем перейти к обработке следующего сообщения, функция окна закончит обработку предыдущего. Это справедливо для асинхронных сообщений, но может быть нарушено для синхронных.

Функция окна – вторая ключевая точка каждого Windows приложения. От того, обработчики каких сообщений вы предусмотрите в оконной процедуре, зависит поведение вашего приложения. Можно сказать, что одно Windows приложение отличается от другого только наполнением оконных функций. Вся "жизнь" вашей программы определяется функцией окна.

Как уже было сказано, функция окна является функцией обратного вызова, что подчеркивается макросом **CALLBACK** в ее заголовке. Параметры функции *всегда* соответствуют первым четырем полям структуры **MSG**. Заметим, что дескриптор окна передается

первым параметром, следовательно нет необходимости запоминать его в глобальной переменной во время создания окна.

Первый параметр функции окна подчеркивает тот факт, что данная функция будет обслуживать *все* окна, созданные на базе данного класса, отличая одно окно от другого по его уникальному дескриптору.

Наша первая функция окна весьма проста – она передает все поступающие сообщения на обработку по умолчанию, вызывая функцию API **`DefWindowProc()`**. Однако *каждое*, даже самое простое Windows приложение должно обработать сообщение **`WM_DESTROY`**, которое поступает в очередь сообщений, когда Windows разрушает ваше окно, что, в свою очередь, является следствием нажатия пользователем кнопки закрытия окна. Стандартным ответом вашей оконной функции должно быть обращение к API функции **`PostQuitMessage()`**, которая помещает последнее для приложения сообщение **`WM_QUIT`** в очередь сообщений. Это, как мы уже знаем, приводит к завершению цикла в **`WinMain()`**. Параметр функции **`PostQuitMessage()`** обычно не используется и равен нулю.

Таким образом, для создания Windows приложения вы обязательно должны написать две функции – **`WinMain()`** и **`WinProc()`.**

Функция **`WinMain()`** проводит регистрацию класса окна, создает главное окно приложения и запускает цикл обработки сообщений.

Оконная функция **`WinProc()`** определяет поведение приложения и должна обрабатывать сообщение **`WM_DESTROY`**, как минимум, чтобы завершить цикл обработки сообщений в **`WinMain()`.**

Обработка сообщений в оконной функции.

Как было отмечено в предыдущем разделе, поведение Windows приложения определяется обработчиками сообщений в функции окна. О том, какие основные сообщения получает оконная функция, как их нужно обрабатывать, и будет посвящен данный раздел. Нет никакой необходимости обрабатывать *абсолютно все* сообщения Windows. Вы должны выявить главные для вашего конкретного приложения и сконцентрироваться на них.

Каждое получаемое окном сообщение идентифицируется номером, который содержится во втором параметре оконной процедуры, а дополнительная информация – в третьем и четвертом.

Если оконная функция обрабатывает сообщение, то ее возвращаемым значением, как правило, будет нулевое значение. Именно так нужно поступать, если в излагаемом ниже материале не приводится дополнительных сведений о коде возврата из обработчика.

Все сообщения, не обрабатываемые оконной процедурой, должны передаваться функции Windows **`DefWindowProc()`**. Такой механизм позволяет Windows проводить обработку сообщений окна совместно с приложением. При этом, значение, возвращаемое функцией **`DefWindowProc()`**, должно быть возвращаемым значением оконной процедуры.

Типичный вид функции окна мы рассмотрели в предыдущем разделе.

Напомним, что функция окна получает сообщение из двух источников: из цикла обработки сообщений и непосредственно от Windows. Из цикла обработки сообщений поступают все асинхронные сообщения, например, сообщения клавиатурного ввода, сообщения о перемещениях и нажатиях клавиш мыши, а также сообщения событий таймера. Непосредственно Windows вызывает функцию окна в виде реакции на синхронные сообщения.

Создание окна WM_CREATE

Это первое сообщение, которое получает функция окна. Его особенность в том, что оконная функция получает **WM_CREATE** только *один раз до возврата* управления из функции **CreateWindowEx()**. Это означает, что окно создано не до конца и, следовательно, очередь сообщений еще отсутствует. Отсюда следует вывод, что сообщение **WM_CREATE** является синхронным и посылается, минуя цикл обработки сообщений, который также еще не запущен. Поскольку сообщение посылается до того, как окно становится видимым, вы можете использовать его для какого-либо "досоздания" окна, например, для создания дочерних окон. В этом случае вам может понадобиться информация из структуры **CREATESTRUCT**, указатель на которую передается через параметр **lParam.**

Если обработчик сообщения возвращает (-1), создание главного окна прекращается. Если обработчик сообщения возвращает 0, создание окна продолжается, и функция **CreateWindowEx()** возвращает дескриптор нового окна. В этом случае, лучшим решением выхода из **WM_CREATE** будет передача управления на обработку по умолчанию.

Определение размера окна WM_SIZE

Это второе сообщение, которое получает функция окна после создания. Кроме того, оно поступает всякий раз, когда пользователь уже изменил размеры окна (в отличие от **WM_SIZING**, которое сигнализирует о том, что пользователь меняет размеры окна). Сообщение передается через очередь сообщений.

Windows приложение, как правило, не знает заранее размеров своих окон. К тому же в любой момент эти размеры можно изменить, например, при помощи мыши. Поэтому приложение должно быть готовым к такому изменению.

Дополнительные параметры сообщения несут информацию о новых размерах окна и о способе, которым оно изменило размер, а именно: **wParam** сообщает о способе изменения размеров. Особый интерес представляют значения **SIZE_MAXIMIZED** и **SIZE_MINIMIZED**, что дает возможность организовать собственную обработку ситуаций, когда окно распахнуто на весь экран или минимизировано. Младшее слово параметра **lParam** сообщает о новой ширине клиентской части окна, а старшее слово –о новой высоте.

Типичное использование **WM_SIZE** – посылка окном самому себе сообщения о перерисовке только измененной части окна. Как это сделать – мы рассмотрим в следующем сообщении.

Для того, чтобы автоматически перерисовывалась *вся* клиентская часть окна при изменении его размеров, при регистрации класса окна определите поле стилей: **wc.style = CS_HREDRAW | CS_VREDRAW;**

Нужно заметить, что узнать текущие размеры клиентской части окна в любом обработчике можно вызвав функцию API:

GetClientRect (HWND hWnd, RECT* lpRect)

которая заполняет поля структуры **RECT** для окна с дескриптором **hWnd.**

Отображение содержимого окна WM_PAINT

Обработка сообщения **WM_PAINT** крайне важна для Windows программирования. Нужно помнить, что в мультипрограммной операционной системе в любой момент может потребоваться перерисовка любого, даже неактивного окна. Проще всего это можно сделать, сосредоточив всю работу по отрисовке в одном месте, которым и является обработчик сообщения **WM_PAINT**. Это сообщение сигнализирует окну, что вся его рабочая область или некоторая ее часть становится недействительной (invalid), и ее следует перерисовать.

Здесь нужно пояснить, что окно разделяется на неклиентскую и рабочую (клиентскую) области. К неклиентской части относится заголовок окна и его обрамление. Перерисовка этой области – забота Windows, поскольку операционная система "знает" о стилях каждого окна, а они то и определяют, что нужно нарисовать в неклиентской части. Вся внутренняя часть окна относится к клиентской области, вот ее то перерисовку вы и должны запрограммировать в обработчике **WM_PAINT**, поскольку Windows не может знать, чего хочет каждый программист.

Случаи генерации сообщения WM_PAINT

Перечислим основные ситуации, когда клиентская область становится недействительной.

- При создании окна недействительна вся его рабочая область. Сообщение **WM_PAINT** помещается в очередь сообщений, когда приложение вызывает функцию **UpdateWindow()**.
- Увеличение (но не уменьшение!) размеров окна, в стиле класса которого заданы флаги **CS_HREDRAW** и **CS_VREDRAW**, приводит к тому, что вся рабочая область также становится недействительной. Операционная система вслед за этим посылает в очередь сообщение **WM_PAINT.**
- Пользователь минимизирует окно программы, а затем вновь восстанавливает его до прежнего размера. Операционная система не сохраняет содержимое рабочей области каждого окна, поскольку в графической среде это было бы слишком накладно. Вместо этого Windows объявляет недействительной всю клиентскую область окна, а затем оконная процедура получает сообщение **WM_PAINT** и сама восстанавливает содержимое окна.
- Во время перемещения окон Windows не сохраняет ту часть окна, которая перекрывается другими окнами. Когда же эта часть позже открывается, Windows вновь отмечает эту область как недействительную, а функция окна получает сообщение **WM_PAINT** для восстановления содержимого окна.
- Если при обработке любого сообщения требуется изменить содержимое окна, то приложение может объявить любую область собственного окна недействительной при помощи функции **InvalidateRect()**, а затем сообщить Windows, что необходимо перерисовать эту часть, вызвав функцию **UpdateWindow()**.

Функция **InvalidateRect()** объявлена следующим образом:

```
void InvalidateRect(HWND hwnd, const RECT* lprc, BOOL fErase)
```

Первый параметр – дескриптор окна, для которого выполняется операция. Второй – указатель на структуру типа **RECT**, определяющую прямоугольную область, подлежащую обновлению. Если указатель равен **NULL**, вся клиентская область объявляется недействительной. Третий параметр указывает на необходимость стирания фона окна, если он задан как **TRUE**, фон окна подлежит стиранию.

Особенность сообщения WM_PAINT

Первая особенность сообщения **WM_PAINT** состоит в том, что оно является низкоприоритетным. Операционная система помещает его в очередь сообщений только тогда, когда там нет других необработанных сообщений.

Вторая особенность – сообщение **WM_PAINT** аккумулируется системой. Это значит, что если в окне объявлены несколько областей, подлежащих обновлению, то приложение получает только одно сообщение **WM_PAINT**, в котором определена суммарная недействительная область, охватывающая все указанные части.

Правила обработки WM_PAINT

Обработка сообщения **WM_PAINT** *всегда* начинается с вызова функции **BeginPaint()**, а заканчивается – вызовом функции **EndPaint()**.

Функция **BeginPaint()** имеет прототип:

HDC BeginPaint (HWND hwnd, PAINTSTRUCT* lpPaint)

Первый параметр – дескриптор окна, а второй – указатель на структуру **PAINTSTRUCT**, поля которой содержат информацию, полезную для проведения профессиональной отрисовки в рабочей области окна. Так, поле **RECT rcPaint** представляет прямоугольник, охватывающий все области окна, требующие перерисовки. Ограничив свои действия по рисованию только этой прямоугольной областью, приложение, несомненно, ускоряет процесс перерисовки.

При обработке вызова **BeginPaint()**, Windows обновляет [или *не обновляет*, это зависит от последнего параметра, с которым была вызвана функция **InvalidateRect()** перед этим] фон рабочей области с помощью кисти, которая указывалась при регистрации класса окна. Поле **fErase** структуры **PAINTSTRUCT** указывает произведено ли перекрашивание фона клиентской области или нет. Знание этого может быть полезным, чтобы не делать повторной закраски в обработчике.

Возвращаемым значением **BeginPaint()** является дескриптор контекста устройства. Этот дескриптор необходим для вывода в рабочую область текста и графики. Заметим, что при использовании данного дескриптора контекста устройства приложение не может рисовать вне клиентской области.

Функция **EndPaint()**, принимающая те же параметры, что и **BeginPaint()**, выполняет обязательное освобождение дескриптора контекста устройства. Обработчик **WM_PAINT** заканчивается вызовом этой функции.

Отрисовка вне WM_PAINT

Функции **BeginPaint()** и **EndPaint()** используются *только* в обработчике **WM_PAINT**. Нужно воспользоваться другими функциями Win32 API, если вашему приложению требуется получить контекст устройства клиентской области окна *вне* **WM_PAINT.** Примерный фрагмент такого кода должен выглядеть следующим образом:

```
// Получить контекст
HDC hDC = GetDC(hWnd);
// операции с дескриптором контекста
// и его освобождение
ReleaseDC(hWnd, hDC);
```

После получения контекста и выполнения каких-либо операций, его обязательно нужно освободить посредством функции **ReleaseDC().**

Системные метрики

Метрики системных компонент Windows можно определить при помощи функции

```
int GetSystemMetrics(int nIndex)
```

Единственный аргумент этой функции задает параметр, значение которого необходимо определить. Значение требуемого параметра возвращается данной функцией.

Для определения того или иного компонента в заголовочных файлах Windows имеются константы с префиксом **SM_**. Здесь перечислены некоторые из них:

SM_CXCURSOR , **SM_CYCURSOR** – ширина и высота курсора.

SM_CXICON, **SM_CYICON** – ширина и высота пиктограммы.

SM_CXSCREEN, **SM_CYSCREEN** – разрешение экрана.

SM_CYCAPTION – высота заголовка окна.

SM_CYMENU – высота одной строки в полосе меню.
SM_CYHSCROLL – высота горизонтальной полосы прокрутки.
SM_CXVSCROLL – ширина вертикальной полосы прокрутки.

Определение расположения окна WM_MOVE

При обсуждении сообщения **WM_SIZE** было показано, как следует определять размеры окна. Другая важная задача – определение расположения окна на экране.

При *завершении* перемещения окна его функция получает сообщение **WM_MOVE** (в отличие от сообщения **WM_MOVING**, которое поступает *во время* перемещения), а вместе с ним – новые координаты окна в параметре **lParam**:

```
xPos = LOWORD(lParam);
yPos = HIWORD(lParam);
```

Для окон, имеющих стили **WS_OVERLAPPED** и **WS_POPUP**, координаты отсчитываются от верхнего левого угла экрана. Для окон стиля **WS_CHILD** эти координаты отсчитываются от верхнего левого угла внутренней области родительского окна.

В любом обработчике, а не только в **WM_MOVE**, можно узнать расположение окна на экране с помощью функции

```
GetWindowRect(HWND hWnd, RECT* lpRect)
```

которая заполняет поля структуры **RECT** текущими координатами окна с дескриптором **hWnd**.

2.3. Сканеры и принтеры штрих-кодов

Задание. Создать приложение, в рабочей области окна которого выводится изображение переплетенных полосок. Ширина полосок и расстояние между полосками должно быть равно h. Использовать сообщение WM_SIZING, препятствуя уменьшению размеров окна до минимально заданных.

Пояснения и указания.

Это второе сообщение, которое получает функция окна после создания. Кроме того, оно поступает всякий раз, когда пользователь уже изменил размеры окна (в отличие от **WM_SIZING**, которое сигнализирует о том, что пользователь меняет размеры окна). Сообщение передается через очередь сообщений.

Windows приложение, как правило, не знает заранее размеров своих окон. К тому же в любой момент эти размеры можно изменить, например, при помощи мыши. Поэтому приложение должно быть готовым к такому изменению.

Дополнительные параметры сообщения несут информацию о новых размерах окна и о способе, которым оно изменило размер, а именно: **wParam** сообщает о способе изменения размеров. Особый интерес представляют значения **SIZE_MAXIMIZED** и **SIZE_MINIMIZED**, что дает возможность организовать собственную обработку ситуаций, когда окно распахнуто на весь экран или минимизировано. Младшее слово параметра **lParam** сообщает о новой ширине клиентской части окна, а старшее слово –о новой высоте.

Типичное использование **WM_SIZE** – посылка окном самому себе сообщения о перерисовке только измененной части окна. Как это сделать – мы рассмотрим в следующем сообщении.

Для того, чтобы автоматически перерисовывалась *вся* клиентская часть окна при изменении его размеров, при регистрации класса окна определите поле стилей: **`wc.style = CS_HREDRAW | CS_VREDRAW;`**

Нужно заметить, что узнать текущие размеры клиентской части окна в любом обработчике можно вызвав функцию API:

```
GetClientRect (HWND hWnd, RECT* lpRect)
```

которая заполняет поля структуры **`RECT`** для окна с дескриптором **`hWnd`**.

2.4. Магнитные карты

Задание. Создать приложение, в рабочей области окна которого выводится строка "График функции Cos(x) для x от -2*π до 2* π" и изображение этого графика. При изменении размеров окна размер изображения графика должно масштабироваться.

Пояснения и указания.

Отображение содержимого окна WM_PAINT

Обработка сообщения **`WM_PAINT`** крайне важна для Windows программирования. Нужно помнить, что в мультипрограммной операционной системе в любой момент может потребоваться перерисовка любого, даже неактивного окна. Проще всего это можно сделать, сосредоточив всю работу по отрисовке в одном месте, которым и является обработчик сообщения **`WM_PAINT`**. Это сообщение сигнализирует окну, что вся его рабочая область или некоторая ее часть становится недействительной (invalid), и ее следует перерисовать.

Здесь нужно пояснить, что окно разделяется на неклиентскую и рабочую (клиентскую) области. К неклиентской части относится заголовок окна и его обрамление. Перерисовка этой области – забота Windows, поскольку операционная система “знает” о стилях каждого окна, а они то и определяют, что нужно нарисовать в неклиентской части. Вся внутренняя часть окна относится к клиентской области, вот ее то перерисовку вы и должны запрограммировать в обработчике **`WM_PAINT`**, поскольку Windows не может знать, чего хочет каждый программист.

Случаи генерации сообщения WM_PAINT

Перечислим основные ситуации, когда клиентская область становится недействительной.

- При создании окна недействительна вся его рабочая область. Сообщение **`WM_PAINT`** помещается в очередь сообщений, когда приложение вызывает функцию **`UpdateWindow()`**.
- Увеличение (но не уменьшение!) размеров окна, в стиле класса которого заданы флаги **`CS_HREDRAW`** и **`CS_VREDRAW`**, приводит к тому, что вся рабочая область также становится недействительной. Операционная система вслед за этим посылает в очередь сообщение **`WM_PAINT`**.
- Пользователь минимизирует окно программы, а затем вновь восстанавливает его до прежнего размера. Операционная система не сохраняет содержимое рабочей области каждого окна, поскольку в графической среде это было бы слишком накладно. Вместо этого Windows объявляет недействительной всю клиентскую область окна, а затем оконная процедура получает сообщение **`WM_PAINT`** и сама восстанавливает содержимое окна.
- Во время перемещения окон Windows не сохраняет ту часть окна, которая перекрывается другими окнами. Когда же эта часть позже открывается, Windows вновь отмечает

эту область как недействительную, а функция окна получает сообщение **WM_PAINT** для восстановления содержимого окна.

– Если при обработке любого сообщения требуется изменить содержимое окна, то приложение может объявить любую область собственного окна недействительной при помощи функции **InvalidateRect()**, а затем сообщить Windows, что необходимо перерисовать эту часть, вызвав функцию **UpdateWindow()**.

Функция **InvalidateRect()** объявлена следующим образом:

```
void InvalidateRect(HWND hwnd, const RECT* lprc, BOOL fErase)
```

Первый параметр – дескриптор окна, для которого выполняется операция. Второй – указатель на структуру типа **RECT**, определяющую прямоугольную область, подлежащую обновлению. Если указатель равен **NULL**, вся клиентская область объявляется недействительной. Третий параметр указывает на необходимость стирания фона окна, если он задан как **TRUE**, фон окна подлежит стиранию.

Особенность сообщения WM_PAINT

Первая особенность сообщения **WM_PAINT** состоит в том, что оно является низкоприоритетным. Операционная система помещает его в очередь сообщений только тогда, когда там нет других необработанных сообщений.

Вторая особенность – сообщение **WM_PAINT** аккумулируется системой. Это значит, что если в окне объявлены несколько областей, подлежащих обновлению, то приложение получает только одно сообщение **WM_PAINT**, в котором определена суммарная недействительная область, охватывающая все указанные части.

Правила обработки WM_PAINT

Обработка сообщения **WM_PAINT** *всегда* начинается с вызова функции **BeginPaint()**, а заканчивается – вызовом функции **EndPaint()**.

Функция **BeginPaint()** имеет прототип:

```
HDC BeginPaint (HWND hwnd, PAINTSTRUCT* lpPaint)
```

Первый параметр – дескриптор окна, а второй – указатель на структуру **PAINTSTRUCT**, поля которой содержат информацию, полезную для проведения профессиональной отрисовки в рабочей области окна. Так, поле **RECT rcPaint** представляет прямоугольник, охватывающий все области окна, требующие перерисовки. Ограничив свои действия по рисованию только этой прямоугольной областью, приложение, несомненно, ускоряет процесс перерисовки.

При обработке вызова **BeginPaint()**, Windows обновляет [или *не обновляет*, это зависит от последнего параметра, с которым была вызвана функция **InvalidateRect()** перед этим] фон рабочей области с помощью кисти, которая указывалась при регистрации класса окна. Поле **fErase** структуры **PAINTSTRUCT** указывает произведено ли перекрашивание фона клиентской области или нет. Знание этого может быть полезным, чтобы не делать повторной закраски в обработчике.

Возвращаемым значением **BeginPaint()** является дескриптор контекста устройства. Этот дескриптор необходим для вывода в рабочую область текста и графики. Заметим, что при использовании данного дескриптора контекста устройства приложение не может рисовать вне клиентской области.

Функция **EndPaint()**, принимающая те же параметры, что и **BeginPaint()**, выполняет обязательное освобождение дескриптора контекста устройства. Обработчик **WM_PAINT** заканчивается вызовом этой функции.

Отрисовка вне WM_PAINT

Функции **BeginPaint()** и **EndPaint()** используются *только* в обработчике **WM_PAINT**. Нужно воспользоваться другими функциями Win32 API, если вашему приложению требуется получить контекст устройства клиентской области окна *вне* **WM_PAINT**. Примерный фрагмент такого кода должен выглядеть следующим образом:

```
// Получить контекст
HDC hDC = GetDC(hWnd);
// операции с дескриптором контекста
// и его освобождение
ReleaseDC(hWnd, hDC);
```

После получения контекста и выполнения каких-либо операций, его обязательно нужно освободить посредством функции **ReleaseDC()**.

2.5. RFID системы

Задание. Создать приложение, в окне которого при нажатии клавиш-стрелок выводится маршрут, задаваемый пользователем. После нажатия клавиши "Enter" по заданному маршруту определяется кратчайший путь, который выводится другим цветом.

Пояснения и указания.

Архитектура Windows, основанная на сообщениях, идеальна для работы с клавиатурой. Приложение узнает о нажатиях клавиш посредством сообщений, которые посылаются оконной процедуре. Когда пользователь нажимает и отпускает клавиши, драйвер клавиатуры передает эту информацию в Windows, которая сохраняет данную информацию в *системной очереди сообщений*, а оттуда – в очередь сообщений окна приложения.

Но какого окна? Ответ – окна, имеющего *фокус ввода*. Смысл этого двухступенчатого процесса – сохранение сообщений в системной очереди сообщений и дальнейшая их передача в очередь сообщений приложения – в синхронизации. Если пользователь нажимает клавиши клавиатуры быстрее, чем приложение может обрабатывать поступающую информацию, Windows сохраняет информацию о дополнительных нажатиях клавиш в системной очереди сообщений.

Одно из дополнительных нажатий может быть переключением фокуса ввода на другую программу. Значит информацию о последующих нажатиях следует направлять окну другого приложения.

Именно таким образом операционная система корректно синхронизирует события клавиатуры.

Для отражения различных событий клавиатуры, Windows посылает программе восемь различных сообщений. Приложения вполне могут игнорировать многие из них. Кроме того, в большинстве случаев в этих сообщениях от клавиатуры содержится значительно больше закодированной информации, чем нужно приложению. Залог успешной работы с клавиатурой – это знание того, какие сообщения важны для приложения, а какие нет.

Фокус ввода и активное окно

Клавиатура должна разделяться между всеми приложениями, работающими под Windows. Некоторые приложения могут иметь больше одного окна, и клавиатура должна разделяться между этими окнами в рамках одного и того же приложения. Когда на клавиатуре нажата клавиша, только одна оконная процедура может получить сообщение об этом. Окно, которое в настоящее время получает все клавиатурные сообщения, именуется окном, имеющим *фокус ввода*.

Концепция фокуса ввода тесно связана с концепцией *активного окна*. Окно, имеющее фокус ввода – это либо активное окно, либо дочернее окно активного окна.

Определить активное окно достаточно просто:

– Windows выделяет заголовок активного окна;
– если у активного окна вместо заголовка имеется рамка диалога, то Windows выделяет ее;
– если активное окно минимизировано, то Windows выделяет текст заголовка в панели задач.

Если активное окно минимизировано, то окна с фокусом ввода нет. Windows продолжает слать программе сообщения клавиатуры, но эти сообщения выглядят иначе, чем сообщения, направленные активным и еще не минимизированным окнам.

Можно обрабатывать сообщения **WM_SETFOCUS** и **WM_KILLFOCUS**, чтобы определить какое окно имеет фокус ввода, однако эти сообщения носят чисто информирующий характер. Программный интерфейс Windows содержит две функции, позволяющие узнать или изменить окно, владеющее фокусом ввода, – **GetFocus()** и **SetFocus().**

Генерация клавиатурных сообщений

Сообщения, которые приложение получает от Windows о событиях, относящихся к клавиатуре, различаются на *аппаратные* (keystrokes) и *символьные* (characters). Такое положение соответствует двум представлениям о клавиатуре. Во-первых, клавиатуру можно считать набором клавиш. В клавиатуре имеется только одна клавиша <A>. Нажатие и отпускание этой клавиши представляют собой аппаратные события. Во-вторых, клавиатура также является устройством ввода, генерирующим отображаемые символы. Клавиша <A>, в зависимости от состояния клавиш <Ctrl>, <Shift> и <CapsLock>, может стать источником нескольких символов. Обычно, этим символом является строчная, латинская ‘a’. Если же нажата клавиша <Shift> или установлен режим <CapsLock>, то этим символом является прописная ‘A’. Если же нажата клавиша <Ctrl>, этим символом будет <Ctrl+A>.

Для сочетаний двух аппаратных событий, которые генерируют отображаемые символы, Windows посылает программе и аппаратные, и символьные сообщения. Некоторые клавиши не генерируют символов. Это такие клавиши, как клавиши переключения, функциональные клавиши, клавиши управления курсором и специальные клавиши, например, <Insert> и <Delete>. Для таких клавиш Windows вырабатывает только аппаратные сообщения.

Аппаратные сообщения

Когда пользователь нажимает клавишу, Windows помещает в очередь окна, имеющего фокус ввода, либо сообщение **WM_KEYDOWN,** либо сообщение **WM_SYSKEYDOWN**. Когда же клавиша отпускается, Windows посылает в очередь либо сообщение **WM_KEYUP,** либо сообщение **WM_SYSKEYUP**. Сообщения **WM_SYS...** вырабатываются при нажатии клавиш в сочетании с клавишей <Alt>.

Итак, **WM_KEYDOWN** и **WM_KEYUP** это несистемные аппаратные сообщения , а системные – **WM_SYSKEYDOWN** и **WM_SYSKEYUP.**

Обычно сообщения о нажатии и отпускании появляются парами. Однако, если пользователь оставит клавишу нажатой так, чтобы включился автоповтор, то Windows посылает оконной процедуре серию сообщений **WM_KEYDOWN** или **WM_SYSKEYDOWN** и *только одно* сообщение **WM_KEYUP** или **WM_SYSKEYUP**, когда в конце концов клавиша будет отпущена. Аппаратные сообщения клавиатуры являются асинхронными сообщениями, и становятся в очередь сообщений окна.

Системные аппаратные сообщения

WM_SYSKEYDOWN и **WM_SYSKEYUP** – системные аппаратные сообщения более важны для Windows, чем для приложений. Эти сообщения генерируются при нажатии клавиш в сочетании с клавишей <Alt>. Они вызывают опции меню программы или системного

меню, или используются для системных функций, таких как смена активного приложения по <Alt+Tab>.

Программы обычно игнорируют системные аппаратные сообщения и передают их в функцию **DefWindowProc()**. Оконная процедура в конце концов получает другие сообщения, являющиеся результатом этих аппаратных сообщений клавиатуры, например, выбор меню.

Если приложение проводит обработку системных аппаратных сообщений, после этого *обязательно* передавайте их в **DefWindowProc()**, чтобы Windows могла использовать эти сообщения в своих целях. Если этого не сделать, то происходит *полная блокировка всех операций* с клавишей <Alt>.

Несистемные аппаратные сообщения

Для клавиш, которые нажимаются и отпускаются без участия клавиши <Alt>, генерируются несистемные сообщения **WM_KEYDOWN** и **WM_KEYUP**. Приложение может использовать или не использовать эти сообщения клавиатуры. Это не влияет на работу операционной системы, поскольку Windows их полностью игнорирует.

Битовые поля параметра lParam

Для всех – системных и несистемных – аппаратных сообщений клавиатуры 32-х разрядная переменная **lParam**, передаваемая в оконную процедуру, состоит из следующих полей:

Биты	Значение
00-15	счетчик повторений (repeat count); равен числу нажатий клавиши. В большинстве случаев 1. Однако, если клавиша остается нажатой, а оконная процедура недостаточно быстра, чтобы обрабатывать эти сообщения в темпе автоповтора, то Windows объединяет несколько аппаратных сообщений о нажатии клавиши в одно сообщение и соответственно увеличивает счетчик повторений. При отпускании клавиши счетчик всегда 1.
16-23	скан-код OEM (Original Equipment Manufacturer scan code); код клавиатуры, генерируемым аппаратурой компьютера. Приложения Windows обычно игнорируют скан-код OEM.
24	флаг расширенной клавиатуры (extended key flag); устанавливается в 1, если сообщение клавиатуры появилось при работе с дополнительными клавишами расширенной клавиатуры IBM. Расширенная клавиатура имеет функциональные клавиши сверху и отдельную комбинированную область клавиш управления курсором и цифр.
25-28	зарезервировано;
29	код контекста (context code); Устанавливается в 1, если нажата клавиша <Alt>. Этот разряд всегда равен 1 для системных аппаратных сообщений и 0 для несистемных аппаратных сообщений клавиатуры за исключением одного случая: если активное окно минимизировано, оно не имеет фокуса ввода, при этом *все нажатия* клавиш вырабатывают сообщения **WM_SYSKEYDOWN** и **WM_SYSKEYUP**.
30	флаг предыдущего состояния клавиши; 1 если клавиша была нажата перед этим;
31	флаг состояния клавиши (transation state).

Виртуальные коды клавиш

Гораздо более важным параметром аппаратных сообщений клавиатуры является параметр **wParam**. В этом параметре содержится *виртуальный код клавиши* (virtual key code), идентифицирующий нажатую или отпущенную клавишу.

Параметр **wParam** содержит код виртуальной клавиши, соответствующей нажатой клавише. Именно этот параметр используется приложением для идентификации клавиши. Код виртуальной клавиши не зависит от аппаратной реализации клавиатуры. Коды виртуальных клавиш имеют символьные обозначения, определенные в заголовочных файлах Windows, и имеют префикс **VK_**.

Символьные сообщения

Для того, чтобы символьные сообщения клавиатуры появились в очереди сообщений окна нужно дополнить цикл обработки сообщений приложения:

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

Функция **GetMessage()** по-прежнему заполняет поля структуры **msg** данными следующего сообщения из очереди, а **DispatchMesage()** отправляет **msg** на обработку в оконную процедуру. Между ними находится функция **TranslateMessage()**, преобразующая аппаратные сообщения клавиатуры в символьные сообщения в соответствии с состоянием драйвера клавиатуры, а также положением управляющих клавиш <Shift> и <CapsLock>. Именно благодаря **TranslateMessage()**, в очереди сообщений появляются символьные сообщения, а приложение сможет отличить <ф> от <А> при активном драйвере кириллицы. Символьное сообщение *всегда* будет следующим, после сообщения о нажатии клавиши, которое функция **GetMessage()** извлечет из очереди.

Существует четыре вида символьных сообщений – **WM_CHAR** и **WM_DEADCHAR** относятся к несистемным и приходят вслед за **WM_KEYDOWN**. Сообщения **WM_SYSCHAR** и **WM_SYSDEADCHAR** являются системными и вызваны появлением сообщения **WM_SYSKEYDOWN.**

Обработка сообщения WM_CHAR

В большинстве случаев программы для Windows могут игнорировать все клавиатурные сообщения за исключением **WM_CHAR**. Параметр **lParam**, передаваемый в оконную процедуру, является таким же, как и параметр **lParam** аппаратного сообщения, из которого сгенерировано символьное сообщение. Параметр **wParam** – это код символа ANSI.

Наиболее типичный обработчик сообщения **WM_CHAR** выглядит следующим образом:

```
case WM_CHAR:
       switch((char)wParam) {
       case 'b': // получен символ Backspace
              //....
              break;
              //....
       case '\t':
              //....
              break;
       case 'A':
              //....
              break;
       case 'a':
              //....
              break;
       //....
       }
return 0;
```

Определение состояния управляющих клавиш

Параметры **wParam** и **lParam** аппаратных сообщений клавиатуры ничего не сообщают о состоянии управляющих клавиш <Shift>, <Ctrl>, <Alt> и клавиш переключателей <CapsLock>, <NumLock>, <ScrollLock>. Приложение может получить текущее состояние любой виртуальной клавиши с помощью функции **GetKeyState()**. Например:

```
case WM_KEYDOWN:
// Нажата ли комбинация <Ctrl>+S ?
if (wParam == 'S' &&
   (0x8000 & GetKeyState(VK_CONTROL))) {
      // ваши действия
}
return 0;
```

Функция **GetKeyState()** не отражает состояние клавиатуры в реальном времени. Она позволяет узнать состояние клавиатуры *на момент, когда последнее сообщение от клавиатуры было выбрано из очереди*. Следовательно, ее можно использовать только в обработчиках клавиатурных сообщений. Недостаток? Это обращается преимуществом, потому что **GetKeyState()** обеспечивает возможность получения точной информации, даже если сообщение обрабатывается асинхронно, уже после того, как состояние переключателя было изменено.

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

2.6. Бесконтактные смарт-карты

Задание. Создать приложение, в окне которого выводится траектория движения курсора мыши. Причем: 1) при движении мыши с нажатой левой клавишей выводятся прямоугольники; 2) при движении мыши с нажатой правой клавишей выводятся окружности; 3) при движении без нажатия клавиш выводится символ '*'. Предусмотреть запись в файл текущего трека манипулятора.

Пояснения и указания.

Сообщение **WM_DESTROY** появляется в очереди сообщений одним из последних. Оно показывает, что Windows находится в процессе ликвидации окна в ответ на полученную от пользователя команду. Пользователь вызывает поступление этого сообщения, если нажмет на кнопке закрытия окна, выберет пункт “Закрыть” из системного меню или нажмет комбинацию клавиш Alt+F4.

Функция главного окна стандартно реагирует на это сообщение, вызывая функцию **PostQuitMessage(0)**, которая ставит последнее для приложения сообщение **WM_QUIT** в очередь сообщений. Это заставляет функцию **WinMain()** закончить цикл обработки сообщений и выйти в систему, завершив работу приложения.

2.7. Фискальные регистраторы и POS системы

Задание. Создать приложение, позволяющее при получении сообщения от таймера выводить символ * в случайном месте рабочей области окна с использованием случайного цвета. Необходимо вести обработку сообщений WM_CREATE, WM_DESTROY, WM_PAINT, WM_TIMER.

Пояснения и указания.

Таймер в Windows можно отнести к устройству ввода информации, которое периодически извещает приложение о том, что истек заданный интервал времени. Приложение сообщает операционной системе интервал времени, а затем Windows периодически сигна-

лизирует программе об истечении этого интервала. Это достигается посылкой сообщения **WM_TIMER**.

Случаи применения таймера в Windows:

1. Режим автосохранения – таймер может предложить программе сохранять работу пользователя на диске, когда истекает заданный интервал времени.
2. Поддержка обновления информации – программа может использовать таймер для вывода на экран обновляемой в реальном времени, постоянно меняющейся информации, связанной либо с системными ресурсами, либо с процессом выполнения отдельной задачи.
3. Поиск другого приложения, запущенного из текущего.
4. Многозадачность – хотя Windows является вытесняющей многозадачной средой, иногда самое эффективное решение для программы – как можно быстрее вернуть управление Windows. Если программа должна выполнять большой объем работы, она может разделить задачу на части и обрабатывать каждую часть при получении сообщения от таймера.
5. Задание темпа изменения – графические объекты в играх или окна с результатами в обучающих программах могут нуждаться в задании установленного темпа изменения.

Поскольку приложения Windows получают сообщения **WM_TIMER** из обычной очереди сообщений, приложение не должно беспокоиться о том, что его работа будет прервана внезапным сообщением **WM_TIMER**. В этом смысле таймер похож на клавиатуру и мышь: драйвер обрабатывает асинхронные аппаратные прерывания, а Windows преобразует эти прерывания в регулярные, структурированные, последовательные сообщения.

Сообщения таймера ставятся в очередь сообщений и обрабатываются как все остальные сообщения, т.е. это сообщение не выделяется среди других, наоборот – оно имеет *наименьший приоритет*.

Операционная система Windows *не является операционной системой реального времени*.

Особенности Windows таймера.

1. *Не гарантируется*, что приложение будет получать сообщение **WM_TIMER** точно по истечению заданного интервала, он будет колебаться. Если приложение занято больше чем секунду, то оно *вообще не получит* ни одного сообщения **WM_TIMER** в течение этого времени. Фактически, Windows обрабатывает сообщение **WM_TIMER** во многом так же, как сообщения **WM_PAINT**. Оба эти сообщения имеют низкий приоритет, и приложение получит их, только если в очереди нет других сообщений.

2. Сообщения **WM_TIMER** похожи на сообщения **WM_PAINT** и в другом смысле – Windows *никогда* не хранит в очереди сообщений несколько сообщений **WM_TIMER**. Вместо этого Windows объединяет несколько сообщений таймера в одно. В результате у приложения нет возможности определить число “потерянных” сообщений **WM_TIMER**.

Windows программа может запустить сколько угодно таймеров. Каждый из них характеризуется уникальным – в рамках приложения – идентификатором. Присоединить таймер к Windows окну можно при помощи вызова функции

```
UINT SetTimer(HWND hWnd, UINT nIDEvent, UINT uElapse,
                                  TIMERPROC lpTimerFunc);
```

Второй параметр функции идентифицирует таймер, а третий – задает интервал в миллисекундах. Это значение определяет темп, с которым Windows будет посылать приложению сообщения **WM_TIMER**.

Для остановки потока сообщений от таймера приложение должно вызвать функцию

```
BOOL KillTimer(HWND hWnd, UINT uIDEvent);
```

Вызов **KillTimer()** очищает очередь сообщений от всех необработанных сообщений **WM_TIMER**.

Приложение должно перед завершением программы уничтожить все активные (запущенные) таймеры.

Использование таймера. Первый способ

Если при запуске таймера определить последний параметр функции **SetTimer()** как NULL, это будет сигналом для Windows генерировать сообщения **WM_TIMER**, которые будут обрабатываться *в обычной оконной процедуре* вашего приложения. При этом код может выглядеть следующим образом:

```
#define TIMER_SEC   1
#define TIMER_MIN   2
//. . . . .
SetTimer(hWnd, TIMER_SEC,  1000, NULL);
SetTimer(hWnd, TIMER_MIN, 60000, NULL);
```

Первый параметр функции **SetTimer()** – это дескриптор того окна, чья оконная процедура будет получать сообщения **WM_TIMER**. Вторым параметром является идентификатор таймера. Его значение должно быть больше нуля. Третий параметр – это 32-х разрядное беззнаковое целое, которое задает интервал в миллисекундах. Например, значение 1000 задает генерацию сообщений **WM_TIMER** один раз в секунду.

Когда оконная процедура получает сообщение **WM_TIMER**, значение **wParam** равно значению идентификатора таймера, а **lParam** для данного случая NULL.

Значение параметра **wParam** позволяют отличать передаваемые в оконную процедуру сообщения **WM_TIMER** от разных таймеров. Логика обработки сообщения **WM_TIMER** выглядит примерно так:

```
case  WM_TIMER:
KillTimer(hWnd, wParam); // остановить таймер
switch(wParam) {
       case TIMER_SEC: // раз в секунду
       // ваши действия ...
       // вновь запускает таймер
       SetTimer(hWnd, TIMER_SEC, 1000, NULL);
       break;
       case TIMER_MIN: // один раз в минуту
       // ваши действия ...
       // вновь запускает таймер
       SetTimer(hWnd, TIMER_MIN, 60000, NULL);
       break;
}
return 0;
```

Для того, чтобы установить новое время срабатывания для существующего таймера, следует сначала его остановить функцией **KillTimer()** и вновь запустить при помощи **SetTimer()**.

Использование таймера. Второй способ

При первом способе установки таймера сообщения **WM_TIMER** посылаются в обычную оконную процедуру. С помощью второго способа можно заставить Windows пересылать сообщения другой функции этого же приложения.

Функция, которая будет получать эти таймерные сообщения должна быть функцией обратного вызова (**CALLBACK**). Эта функция приложения, которую, как и оконную процедуру, вызывает Windows. Приложение должно сообщить Windows адрес этой функции, а позже Windows вызывает именно ее, а не оконная процедуру. Как и оконная процедура,

функция обратного вызова должна определяться как **CALLBACK**, поскольку Windows вызывает ее вне кодового пространства программы.

В случае функции обратного вызова для таймера, входными параметрами являются те же параметры, что и параметры оконной процедуры. Однако таймерная функция обратного вызова не имеет возвращаемого в Windows значения.

Допустим, что в качестве имени таймерной функции обратного вызова выбрано имя **TimerProc**, тогда эта функция, которая будет обрабатывать *только* сообщения **WM_TIMER**, должна иметь следующее определение:

```
void CALLBACK TimerProc(HWND hwnd, UINT uMsg,
                        UINT idEvent, DWORD dwTime)
{
       KillTimer(hWnd, idEvent);   // остановить таймер
       if (idEvent == TIMER_SEC) { // раз в секунду
              // ваши действия ...
              // вновь запускает таймер
              SetTimer(hWnd, TIMER_SEC, 1000, TimerProc);
       }
}
```

Первый параметр для **TimerProc()** – дескриптор окна, задаваемый при вызове функции **SetTimer().** Поскольку Windows будет посылать функции **TimerProc()** только сообщения **WM_TIMER,** следовательно, параметр **uMsg** всегда равен **WM_TIMER**. Значение **idEvent** – идентификатор таймера, а значение **dwTime** – системное время.

При использовании функции обратного вызова для обработки сообщений **WM_TIMER**, четвертый параметр функции **SetTimer()** должен быть адресом функции обратного вызова:

```
SetTimer(hWnd, TIMER_SEC, 1000, TimerProc);
```

Второй способ обычно применяется, чтобы разгрузить оконную функцию. На первый взгляд получение значения системного времени тоже относится к преимуществам второго способа. Однако обращение к Win32 API функции

```
GetSystemTime (SYSTEMTIME* lpSystemTime)
```

позволяет решить эту задачу в произвольном месте вашего приложения.

2.8. Банкоматы и платежные терминалы

Создать приложение, позволяющее динамически менять цвет области окна, задавая три его компоненты (R, G, B) при помощи трех полос просмотра.

Пояснения и указания.

Нередко возникает ситуация, когда выводимая в окно информация, допустим, многострочный документ, превышает текущие размеры окна. Стандартным решением этой задачи в Windows программировании является использование полос прокрутки, которые позволяют проводить вертикальный или горизонтальный скроллиг содержимого окна.

Добавить в окно приложения вертикальную или горизонтальную полосу прокрутки не представляет затруднений. Все, что необходимо сделать, это включить битовые маски **WS_VSCROLL** и **WS_HSCROLL** в список стилей окна, создаваемого функцией **CreateWindowEx()**. Следует заметить, что клиентская область окна не включает в себя пространство, занятое полосами прокрутки. Ширина вертикальной полосы и высота горизонтальной полос прокрутки постоянны для конкретного дисплейного драйвера. Эти значения можно получить с помощью функции **GetSystemMetrics().**

Заметим, что после создания окна с полосами прокрутки, они будут присутствовать на экране, но будут бездействовать, т.к. программирование логики их работы – ваша задача.

Заметим также, что Windows обеспечивает только логику работы мыши с полосами прокрутки. Однако у полос прокрутки отсутствует клавиатурный интерфейс. Если приложение желает дублировать клавишами клавиатуры некоторые функции полос прокрутки, оно должно самостоятельно реализовать эту логику при обработке клавиатурных сообщений.

Диапазон и положение полос прокрутки

Каждая полоса прокрутки имеет соответствующий диапазон – два целых числа, отражающих минимальное и максимальное значение, и положение бегунка – его местоположение внутри диапазона. При этом, положение бегунка всегда дискретно. Например, для полосы прокрутки с диапазоном от 0 до 4 имеется пять возможных положений бегунка.

Win32 API содержит две функции, которые позволяют выполнить все операции с полосами прокрутки. Это:

```
GetScrollInfo(HWND hwnd, int fnBar, SCROLLINFO* lpsi)
SetScrollInfo(HWND hwnd, int fnBar, SCROLLINFO* lpsi,
              BOOL fRedraw)
```

Первая из них возвращает, через поля заполненной структуры **SCROLLINFO**, всю необходимую информацию о полосе прокрутки, а вторая – позволяет установить новые значения для полосы.

В качестве первого параметра этих функций задается дескриптор окна, владеющего полосой прокрутки. Параметр **fRedraw** устанавливается в **TRUE**, если необходимо, чтобы Windows перерисовала полосы в соответствии с новыми значениями. Параметр **fnBar** указывает на горизонтальную (**SB_HORZ**) или вертикальную (**SB_VERT**) полосу. Третий параметр **lpsi** является указателем на структуру **SCROLLINFO**, которая имеет следующие поля:

`cbSize` размер структуры в байтах. Вы можете указать `cbSize=sizeof(SCROLLINFO);`

`fMask` флаг, определяющий какие из полей структуры будут заполнены. Так, если `fMask=SIF_RANGE`, будут заполнены значения диапазона полосы, если `fMask=SIF_PAGE`, то поле `nPage`, если `fMask=SIF_POS`, то поле `nPos`. Можно комбинировать значения, принимаемые флагом, для того, чтобы выбрать (или установить) интересующую информацию за одно обращение к функции.

`nMin` минимальное значение диапазона полосы.

`nMax` максимальное значение диапазона полосы.

`nPage` объем информации, укладывающийся на одной странице.

`nPos` текущее положение бегунка. `nMin <= nPos <= nMax`.

`nTrackPos` текущее положение бегунка при протаскивании его мышью.

При создании полосы Windows, по умолчанию, устанавливает минимальное значение диапазона в 0, а максимальное – в 100.

Если вам требуется убрать полосу просмотра у окна, установите `nMin=0` и `nMax=0`.

Значение параметра `nPage` влияет на размер бегунка полосы, что вы можете наблюдать в ряде современных Windows приложений. Чтобы этого достичь в вашем приложении, рассчитайте сколько, например, строк документа размещается в видимой части окна и установите это значение вызовом **SetScrollInfo()**. Windows определит сколько страниц укладывается во всем документе и, в соответствии с этим, установит размер бегунка – чем меньше размер, тем больше исходный документ. Если хотите сохранить постоянные размеры бегунка при любом объеме документа, установите `nPage=0`.

Прежде, чем перейти к обработке сообщений полос просмотра, укажем сферы ответственности Windows и приложения при поддержке полос прокрутки.

Сферы ответственности Windows:

1. Управление логикой работы мыши с полосой прокрутки.
2. Обеспечение временной “инверсии цвета” при нажатии на кнопку мыши в полосе прокрутки.
3. Перемещение бегунка в соответствие с тем, как внутри полосы прокрутки его перемещает пользователь с помощью мыши.
4. Отправка сообщений полосы прокрутки в оконную процедуру окна, содержащего полосу.

Сферы ответственности приложения по поддержке полос покрутки:

1. Инициализация диапазона полосы прокрутки.
2. Обработка сообщений полосы прокрутки.
3. Обновление положения бегунка полосы прокрутки.

Сообщения полос прокрутки

Windows посылает оконной процедуре сообщения **`WM_VSCROLL`** (для вертикальной полосы) и **`WM_HSCROLL`** (для горизонтальной), когда пользователь щелкает мышью на полосе прокрутки или перетаскивает бегунок. Параметры этих сообщений:

```
nScrCode = LOWORD(wParam);
nPos     = HIWORD(wParam);
```

Следует отметить, что при работе с оконными полосами прокрутки параметр **`lParam`** игнорируется, поскольку он используется *только* для полос прокрутки как элементов управления.

Для Win32 приложений также следует игнорировать старшее слово параметра **`wParam`**, представляющего значение положения бегунка. Его нужно получать через функцию **`GetScrollInfo()`**.

Оставшееся младшее слово параметра **`wParam`** данных сообщений показывает какие действия мышью осуществляются на полосе прокрутки. Этот код операций может принимать следующие значения для вертикальной полосы:

`SB_TOP` скроллиг к началу документа.
`SB_BOTTOM` скроллиг к концу документа.
`SB_LINEUP` прокрутка на одну строку вверх.
`SB_LINEDOWN` прокрутка на одну строку вниз.
`SB_PAGEUP` скроллиг на одну страницу вверх.
`SB_PAGEDOWN` скроллиг на одну страницу вниз.
`SB_THUMBTRACK` протаскивание бегунка мышью.

И значения для горизонтальной полосы:

`SB_LEFT` скроллиг на одну страницу влево.
`SB_RIGHT` скроллиг на одну страницу вправо.
`SB_LINELEFT` прокрутка на одну строку влево.
`SB_LINERIGHT` прокрутка на одну строку вправо.
`SB_THUMBTRACK` протаскивание бегунка мышью.

Алгоритм обработчика прокрутки должен включать обновление положения ползунка в соответствии с новым значением, расчет отображаемой части документа и перерисовку клиентской области окна.

Обработкик будет работать значительно эффективнее, если воспользоваться функцией **`ScrollWindowEx()`**, позволяющей осуществлять быстрое перемещение прямоугольной части клиентской области в пределах окна.

2.9. Видеонаблюдение

Задание. Создать приложение, которое в качестве главного окна приложения использует диалоговую панель, выполняющую функции простейшего калькулятора.

Пояснения и указания.

Меню – это наиболее консервативная и, одновременно, наиболее важная часть пользовательского интерфейса. Трудно найти Windows приложение, у которого отсутствует меню, – это считается “дурным тоном” у разработчиков.

Обычно выделяют четыре вида меню:

1. Главное меню окна или меню верхнего уровня (*top-level menu*). Это меню представляет собой горизонтальную строку, которая расположена непосредственно под заголовком окна и состоит из нескольких элементов.

2. Подменю (*submenu*) появляется под главным меню при выборе одного из его элементов.

3. Плавающее меню (*floating menu*). Оно не связано с главным меню, и может быть создано в любом месте экрана как независимое всплывающее меню.

4. У подавляющего большинства главных окон приложений в левой части заголовка находится пиктограмма. Щелчок мышью по ней приводит к появлению системного меню (*system menu*), которое похоже на подменю главного меню приложения. Обычно системное меню всех приложений одинаково, с его помощью можно минимизировать или максимизировать окно приложения, перемещать его и прочее. Однако приложение имеет возможность изменить системное меню, дополняя его новыми строками, или удаляя существующие.

Возможные состояния пунктов меню

1. При выборе пункта меню, строка инвертирует цвет. Это применимо к пунктам меню всех видов.

2. Пункты всплывающих меню могут находиться в состоянии *отмечено* (checked), при этом слева от текста пункта меню выставляется специальная метка. Эта метка позволяет пользователю узнать о том, какие опции программы выбраны из этого меню. Пункты главного меню *не могут быть отмечены*.

3. Пункты меню могут находиться в состоянии *активно* (enabled), *неактивно* (disabled) и *недоступно* (grayed). Имейте в виду, что пункты, помеченные как активные или неактивные, выглядят для пользователя одинаково, а недоступный пункт меню выводится в сером цвете.

Только при выборе *активных* пунктов меню Windows генерирует сообщения, поступающие в функцию окна, содержащего это меню.

Итак, каждый пункт меню определяется тремя характеристиками:

1. Внешний вид пункта меню, другими словами – то, что будет отображено в меню. Это может быть либо строка текста, либо графический, точнее битовый образ.

2. Атрибут пункта меню, который определяет, является ли данный пункт активным, недоступным или отмеченным.

3. Уникальный числовой идентификатор, который Windows посылает в функцию окна, когда пользователь выбирает данный пункт меню.

Сообщения от пунктов меню

При выборе пунктов меню Windows генерирует сообщение **WM_COMMAND**. Это сообщение можно назвать сообщением пользовательского интерфейса, поскольку оно посылается всякий раз, когда пользователь что-то выбирает, что-то меняет и так далее. В общем случае это сообщение имеет параметры:

```
WM_COMMAND
wEvent  = HIWORD(wParam);
wID     = LOWORD(wParam);
hWndCtl = (HWND)lParam;
```

Параметр **wEvent** именуется нотификационным кодом или кодом извещения. Для пунктов меню он *всегда* равен нулю. Параметр **wID** – это уникальное числовое значение, ассоциированное с каждым элементом. В случае меню, он содержит идентификатор пункта меню. Параметр **hwndCtl** представляет собой дескриптор окна элемента управления. Для пунктов меню он *всегда* равен NULL.

При выборе пунктов системного меню генерируется сообщение **WM_SYSCOMMAND**, которое имеет такие же параметры.

Напомним еще раз, что сообщение **WM_COMMAND** поступает в функцию окна, когда пункт меню *уже выбран*. Однако решаемая вами задача может потребовать знания, какой пункт меню *выбирается* пользователем в настоящий момент. Допустим, вы хотите динамически менять внешний вид пункта меню, когда пользователь наводит на него курсор мыши. В этом случае функция окна должна обрабатывать сообщение **WM_MENUSELECT**, которое поступает до **WM_COMMAND** и имеет параметры:

```
Item   = LOWORD(wParam);
fFlags = HIWORD(wParam);
hMenu  = (HMENU)lParam.
```

Параметры **Item** и **hMenu** соответствуют идентификатору и дескриптору выбираемого пункта меню, а значение **fFlags** содержит флаги состояния этого пункта.

Создание главного меню приложения

Добавление меню к программе – сравнительно простая задача Windows программирования. Структура меню достаточно просто определяется в описании ресурсов. Каждому пункту меню присваивается числовой идентификатор. Обеспечение уникальности идентификаторов берут на себя современные редакторы ресурсов, вам остается придумать мнемонические имена этим идентификаторам, например **ID_EXIT.**

Для создания меню применяются три метода:

1. Шаблон меню создается в файле ресурсов приложения, а затем загружается при создании главного окна приложения. Этот способ подходит для статических меню, неменяющихся или незначительно меняющихся в процессе работы программы.

2. Меню без шаблона создается динамически только при помощи функций Win32 API. Этот способ подходит для приложений, существенно меняющих внешний вид меню, когда разработать подходящий шаблон заранее не представляется возможным.

3. Шаблон меню подготавливается непосредственно в оперативной памяти и с помощью специальных функций подключается к приложению.

Мы, преимущественно, будем использовать первый способ. Как было сказано, прежде чем подключить меню к приложению, его нужно загрузить из ресурсов. Последовательность действий может быть такой:

```
HMENU hMenu = LoadMenu(g_hInst, "Main_Menu");
HWND   hWnd = CreateWindowEx(
       WS_EX_OVERLAPPEDWINDOW,
       szClass, szTitle, WS_OVERLAPPEDWINDOW,
       0, 0, 100, 100,
       NULL, hMenu, g_hInst, NULL);
if (!hWnd) return -1;
```

Заметим, что при указании ресурсов приложения чаще используются числовые значения, идентифицирующие ресурсы, а не строковые описания, например, **IDR_MAIN_MENU.** В этом случае следует использовать макрос **MAKEINTRESOURCE()** для преобразования идентификатора ресурса в строковое описание. Пример загрузки меню выглядит так:

```
HMENU hMenu = LoadMenu(g_hInst,
              MAKEINTRESOURCE(IDR_MAIN_MENU));
```

Обработчик сообщений меню несложен и может выглядеть следующим образом:

```
case WM_COMMAND: {
    int wID = LOWORD(wParam);
    switch (wID) {
       case ID_EXIT :
           DestroyWindow(hWnd);
           break;
    }
    return 0;  }
```

Функции для работы с меню

1. Если окно **hWnd** приложения имеет главное меню, то получить его дескриптор можно с помощью функции

```
HMENU hMenuMain = GetMenu(hWnd);
```

Затем можно получить дескрипторы всех подменю, вызывая функцию

```
HMENU hSubMenu = GetSubMenu(hMenuMain, nPos);
```

Здесь целочисленное значение **nPos** изменяется от 0 до **nMenu**-1, где переменная **nMenu** обозначает количество подменю. Узнать это значение можно с помощью универсальной функции

```
nMenu = GetMenuItemCount(HMENU hMenu),
```

которая возвращает количество элементов в *любом* меню, а не только в главном.

2. Для полной замены меню у главного окна приложения нужно воспользоваться функцией

```
BOOL SetMenu (HWND hWnd, HMENU hMenuNew),
```

которая позволяет прикрепить к окну **hWnd** новое меню с дескриптором **hMenuNew.**

В этом случае “старое” меню требуется разрушить вызовом функции **DestroyMenu(hMenu)**. Если же меню прикреплено к окну при создании последнего (см. пример предыдущего пункта), Windows автоматически разрушит такое меню при закрытии окна.

3. Изменить состояние пункта меню с идентификатором **uID** можно с помощью вызова функции

```
EnableMenuItem(hMenu, uID, uEnable);
```

Параметр **uEnable** может принимать одно из следующих значений:

MF_DISABLED (недоступно), **MF_ENABLED** (активно) или **MF_GRAYED** (недоступно и отмечено серым цветом). Это значение должно быть объединено с константой **MF_BYCOMMAND**, если параметр **uID** задает идентификатор меню, либо с константой **MF_BYPOSITION**, если **uID** задает номер позиции в меню.

4. Если вы изменяете состояние какого-либо пункта *главного меню* окна **hWnd**, то следует выполнить перерисовку меню вызовом функции

```
DrawMenuBar(hWnd);
```

Перерисовывать подменю не требуется, поскольку это делается Windows автоматически при каждом вызове подменю.

5. Программный интерфейс Win32 содержит две универсальные функции

```
BOOL GetMenuItemInfo(HMENU hMenu, UINT uItem,
       BOOL fByPosition, MENUITEMINFO* pMI)
BOOL SetMenuItemInfo(HMENU hMenu, UINT uItem,
       BOOL fByPosition, MENUITEMINFO* pMI)
```

первая из которых возвращает полную информацию о пункте любого меню, а вторая – позволяет совершить все необходимые операции с этим пунктом. Параметр **uItem** является позицией в меню, если переменная **fByPosition** имеет ненулевое значение, либо идентификатором – в противном случае. Исчерпывающее описание полей структуры **MENUITEMINFO** вы можете найти в любой справочной системе по Win32 API.

2.10. Антикражные ворота

Задание. Создать приложение, являющееся простейшим редактором текста. Приложение позволяет создавать новые файлы, открывать уже существующие, редактировать текст и сохранять его в файле. Для выбора имен файлов используются стандартные диалоговые панели.

Пояснения и указания.

Следующим после меню, часто используемым ресурсом приложения, является диалог. Практически любое стандартное приложение Windows использует диалоговые панели. Однако вначале следует рассмотреть составляющие компоненты диалогов – элементы управления.

В Windows предопределен целый ряд различных элементов управления, таких, как кнопки, редакторы текстов и списки и прочее. Эти элементы управления именуются дочерними окнами управления (*child window control*). Все они создаются на базе предопределенных классов, но разработчик может определить и собственные классы дочерних окон, зарегистрировав их при помощи функции **`RegisterClass()`**.

Вспомним, что дочернее окно, во-первых, определяется стилем **`WS_CHILD`** и *всегда* располагаются на поверхности родительского окна, как бы "прилипая" к нему. Во-вторых, при любом перемещении дочернее окно никогда не выходит за границы родительского окна. Родительское окно может содержать несколько элементов управления, которые будут перемещаться вместе с окном-родителем.

Чтобы родительское окно различало дочерние окна, последние должны иметь уникальный идентификатор и уникальный дескриптор окна.

Итак, достаточно просто создать нужные дочерние окна, указав их размеры, расположение и некоторые другие атрибуты. После этого приложение может взаимодействовать с элементами управления, передавая им и получая от них различные сообщения. При этом каждый элемент управления самостоятельно обрабатывает сообщения мыши и клавиатуры и извещает родительское окно о том, что его состояние изменилось. В этом случае дочернее окно становится для родительского окна устройством ввода. При этом оно инкапсулирует особые действия, связанные с графическим представлением окна на экране, реакцией на пользовательский ввод, и извещения другого окна при вводе любой информации. Приложению нет необходимости беспокоиться о логике обработки мыши этими окнами, или о логике их отрисовки. Все это входит в зону ответственности Windows, а все, что остается приложению – это обрабатывать сообщение **`WM_COMMAND`**. Этим сообщением дочерние окна информируют оконную процедуру о различных событиях.

Win32 API содержит новые элементы управления, которые используют сообщение **`WM_NOTIFY`** для извещения родительского окна.

Создание стандартных элементов управления

Динамически – вне шаблона – создать стандартный элемент управления проще всего с помощью функции **`CreateWindow()`**, используя один из предопределенных классов: "**`button`**", "**`edit`**", "**`static`**", "**`listbox`**", "**`combobox`**" и "**`scrollbar`**".

Например, кнопку можно создать так:

```
HWND hWndBotton = CreateWindow(
       "button", "Отмена",
       dwStyle, x, y, nWidth, nHeight,
       hWndParent,
       (HMENU)nIDctrl,
       g_hInst, NULL);
```

Параметр **`nIDctrl`** – числовой идентификатор окна. Для каждого создаваемого дочернего окна необходимо определить собственный уникальный идентификатор. Родитель-

ское окно будет различать элементы управления по этому параметру, получая сообщение **WM_COMMAND** от *всех* дочерних окон. Дескриптор **hWndParent** – это описатель родительского окна.

Кроме того, нужно указать конкретные параметры стиля **dwStyle** создаваемого окна для более точного определения вида и свойств каждого из элементов управления.

Оконные процедуры для стандартных элементов управления уже включены в состав ядра Windows. Они необходимы для обработки сообщений тех дочерних окон, которые созданы на основе перечисленных классов.

Разрушение элементов управления

При необходимости созданный элемент управления можно удалить функцией **DestroyWindow().** Однако помните, при разрушении родительского окна, Windows *сама удаляет* все его дочерние окна.

Функции для работы с элементами управления

Вызывая функции Win32 API, можно динамически перемещать элементы управления, делать их активным или неактивным, скрывать или, наоборот, отображать в окне.

Переместить элемент управления внутри родительского окна можно с помощью функции **SetWindowPos()**. Эта функция определяет новое расположение и размеры окна в системе координат, связанной с родительским окном.

```
BOOL SetWindowPos (HWND hWnd,
       HWND hWndInsertAfter,
       int X, int Y, int cx, int cy,
       UINT uFlags)
```

Кроме того, данная функция позволяет манипулировать, так называемым z-порядком окон. Эта задача становится актуальной, когда на поверхности родительского окна располагается несколько перекрывающихся дочерних окон, и одно окно нужно “убрать” под другое, или наоборот – вывести его на передний план.

Функция

```
BOOL EnableWindow (HWND hWndChild, BOOL bEnable)
```

делает окно **hWndChild** либо активным, если второй параметр **TRUE**, либо недоступным, если **bEnable** равен **FALSE.**

Функция

```
BOOL IsWindowEnabled (HWND hWndChild)
```

проверяет активно ли окно **hWndChild** в настоящий момент.

Функция

```
BOOL ShowWindow (HWND hWndChild, int nCmdShow)
```

позволяет скрыть (`nCmdShow=SW_HIDE`) или вновь отобразить (`nCmdShow= SW_SHOW`) окно **hWndChild** на поверхности родительского окна.

Укажем еще две вспомогательные функции. Одна из них дает возможность узнать дескриптор дочернего окна, зная его целочисленное значение идентификатора **nID**:

```
HWND GetDlgItem (HWND hWndParent, int nID)
```

а вторая, обратная первой – возвращает идентификатор дочернего окна по его известному дескриптору `hWndChild`:

```
UINT GetDlgCtrlID (HWND hWndChild)
```

Хотя часть “`Dlg`” в именах этих функций относится к окнам диалога, на самом деле – это функции общего назначения.

Сообщения дочерних окон

Если стандартный элемент управления изменяет свое состояние, то функция родительского окна получает сообщение **WM_COMMAND** со следующими параметрами:

```
WM_COMMAND
int  nIDCtrl = LOWORD(wParam);
int  nEvent  = HIWORD(wParam);
HWND hWndCtl = (HWND) lParam;
```

Код уведомления **nEvent** – это дополнительный код, который дочернее окно использует для того, чтобы сообщить родительскому окну более точные сведения о сообщении. Константы, идентифицирующие различные коды уведомления, определены в заголовочных файлах Windows и имеют соответственно следующие префиксы: **BN_** – “button”, **EN_** – ”edit”, **LBN_** – “listbox”, **CBN_** – “combobox” и **SB_** – “scrollbar”.

Параметр **nIDCtrl** – это уникальное числовое значение, которое указывается при создании элемента управления.

Параметр **hwndCtl** представляет собой дескриптор окна элемента управления.

Сообщения родительского окна дочерним окнам

Родительское окно может передавать сообщения дочерним окнам, в ответ на которые это дочернее окно будет выполнять различные действия. Передать можно как обычное оконное сообщение (с префиксом **WM_**), так и специфические для каждого типа элемента управления. Константы, идентифицирующие различные сообщения для дочерних окон управления, определены в заголовочных файлах Windows и имеют соответственно следующие префиксы: **BM_** – “button”, **EM_** – ”edit”, **LB_** – “listbox”, **CB_** – “combobox”.

Напомним, что существует два способа передачи сообщений.

Асинхронный способ – это запись сообщения в очередь приложения. Он основан на использовании функции **PostMessage()**. Эта функция помещает сообщение в очередь сообщений для окна, указанного в параметрах, и сразу же возвращает управление. Позже (асинхронно по времени) записанное при помощи функции **PostMessage()** сообщение будет выбрано и обработано в цикле обработки сообщений.

Синхронный способ – это непосредственная передача сообщения функции дочернего окна, минуя очередь сообщений. Этот метод реализуется функцией **SendMessage().** В отличие от предыдущей функции, функция **SendMessage()** вызывает оконную процедуру и возвращает управление только после выхода из функции дочернего окна.

Для элементов управления синхронный способ является основным, что подчеркивается наличием в Win32 API специально для этого предназначенного варианта функции **SendMessage()**

```
LONG SendDlgItemMessage(
       HWND hWndParent, int nIDitem,
       UINT Msg, WPARAM wParam, LPARAM lParam)
```

которая отправляет сообщение **Msg** дочернему окну с идентификатором **nIDitem**, расположенному на родительском окне **hWndParent.**

Расширенное управление дочерними окнами

Рассмотрим эту возможность на примере решения задачи об изменении цвета фона элемент управления. Когда элемент управления собирается отрисовать свою рабочую область, он посылает процедуре родительского окна соответствующее сообщение с префиксом **WM_CTLCOLOR** (например, для EditBox это сообщение **WM_CTLCOLOREDIT**). Родительское окно на основании этой информации может изменить цвет, который будет использован в оконной процедуре дочернего окна при рисовании. В итоге родительское окно может управлять цветами своих дочерних окон.

Возвращаемым значением для сообщений `WM_CTLCOLOR...` является *дескриптор* на кисть `hBrush`.

Оконные процедуры элементов управления

Как уже отмечалось, функции окон элементов управления находятся в ядре Windows. Однако у разработчика имеется возможность получить адрес этой оконной процедуры с помощью функции **GetWindowLong()**, для которой в качестве второго параметра используется идентификатор **GWL_WNDPROC**. С помощью же функции **SetWindowLong()** можно не только получить адрес оконной процедуры элемента управления, но и установить новую оконную процедуру для него. Этот очень мощный прием именуется *window subclassing*. Он позволяет дополнять код стандартной оконной процедуры новыми возможностями, другими словами, "влезть" в существующие внутри ядра Windows оконные процедуры, обработать некоторые сообщения специфическим для приложения способом, а все остальные сообщения оставить для прежней оконной процедуры. Например, необходимо создать элемент EditBox, который предназначен для ввода только вещественных чисел. Последовательность ваших действий должна быть следующей:

1. Объявить глобальную переменную, как указатель на CALLBACK функцию

```
static WNDPROC OldEditWindowProc;
```

Указатель OldEditWindowProc будет использован для получения адреса оконный процедуры стандартного элемента управления EditBox. Применение глобального класса памяти объясняется тем, что использовать его придется в различных обработчиках.

2. В обработчике **WM_CREATE** родительского окна создается элемент управления, допустим имеющий дескриптор hEdit. После этого, нужно получить адрес "старой" оконный процедуры для hEdit

```
OldEditWindowProc = (WNDPROC)GetWindowLong(
             hEdit, GWL_WNDPROC);
```

и уставить "новую" функцию окна NewEditWindowProc

```
SetWindowLong(hEdit,
             GWL_WNDPROC, (LONG)NewEditWindowProc);
```

3. Новая оконная процедура, конечно же, является функцией обратного вызова и определяется следующим образом:

```
LRESULT CALLBACK NewEditWindowProc (HWND hEdit,
       UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch (uMsg) {
    // ваши обработчики
    case ...
    case WM_DESTROY :
      // при разрушении элемента управления
       // нужно восстановить старый адрес.
       SetWindowLong(hEdit, GWL_WNDPROC,
              (LONG)OldEditWindowProc);
    }
    return CallWindowProc(OldEditWindowProc,
              hEdit, uMsg, wParam, lParam);
}
```

Обратите внимание на то, что обработка по умолчанию для подмененной оконной процедуры выполняется с помощью **CallWindowProc()**, а не **DefWindowProc()**.

3. Задания, выносимые на самостоятельную работу, и рекомендации по выполнению

3.1. Задание. Создать приложение, работающего в двух режимах: кодирование и декодирование содержимого файла. Кодирование и декодирование символов должно происходить при помощи таблицы перекодировки, загружаемой из ресурсов приложения. т.е. ресурсы содержат произвольные данные. Имена файлов выбираются при помощи стандартной диалоговой панели “File Open”. Содержимое закодированного или восстановленного файла выводится в окно редактирования, занимающего всю клиентскую часть окна приложения, и может быть сохранено в файлах *.cod и *.txt соответственно.

Пояснения и указания.

В Win32 API реализован ряд часто употребляемых диалоговых окон, что освобождает разработчика от необходимости их повторной реализации для каждого своего приложения. К этим диалоговым окнам, которые носят название общего использования, относятся следующие диалоги.

Назначение диалога	Вызывающая функция
Получить имя открываемого файла	**BOOL GetOpenFileName (OPENFILENAME* pOfn)**
Получить имя для сохранения файла	**BOOL GetSaveFileName (OPENFILENAME* pOfn)**
Получить текст для поиска	**HWND FindText (FINDREPLACE* pFr)**
Получить текст для замены	**HWND ReplaceText (FINDREPLACE* pFr)**

Вызывающие функции двух последних диалогов возвращают дескриптор окна, а не **BOOL** переменную, как все остальные. Легко догадаться, что эти последние диалоги являются немодальными.

Общие диалоговые окна можно использовать двумя способами.

1. Приложения могут использовать эти диалоги в их стандартном виде, вызывая соответствующую функцию общего диалога как обычную Win32 функцию.

2. Расширенный способ позволяет настраивать общие диалоговые окна, в соответствии с целями приложения, например, заменяя специальным образом шаблоны диалоговых панелей.

В заключение приведем пример программного кода, который дает возможность пользователю выбрать имя файла для сохранения некоторой информации. В этом фрагменте **hWnd** - дескриптор главного окна, **g_hInst** - дескриптор копии приложения, **ofn** - структура **OPENFILENAME**, поля которой используются для настройки диалоговой панели.

```
TCHAR szFullPath[MAX_PATH];
TCHAR szFileName[MAX_PATH];
*szFullPath = 0;
*szFileName = 0;
OPENFILENAME ofn;
// проводим “обнуление” всей структуры
ZeroMemory(&ofn, sizeof(ofn));
ofn.lStructSize = sizeof(ofn);
ofn.hwndOwner   = hWnd;
ofn.hInstance   = g_hInst;
// фильтр для отображения типов файлов
ofn.lpstrFilter = "Текстовые файлы\0*.txt\0
                   Все файлы\0*.*\0\0";
```

```
ofn.nFilterIndex= 1;
ofn.lpstrFile   = szFullPath;
ofn.nMaxFile    = sizeof(szFullPath);
ofn.lpstrFileTitle= szFileName;
ofn.nMaxFileTitle = sizeof(szFileName);
// используем текущий катаог
ofn.lpstrInitialDir=NULL;
// определяем заголовок диалога
ofn.lpstrTitle  = "Сохранить в файле";
// настраиваем флаги диалога
ofn.Flags = OFN_PATHMUSTEXIST|OFN_OVERWRITEPROMPT|
            OFN_HIDEREADONLY |OFN_EXPLORER;
// отображаем диалог
if (GetSaveFileName(&ofn)) {
  // в этой точке имеем:
  // ofn.lpstrFile - полный путь файла
  // ofn.lpstrFileTitle - имя выбранного файла
  // делаем запись содержимого файла
}
```

Функции ядра Windows для работы с файлами

В Win32 API появились принципиально новые 32-х разрядные функции. Именно они рекомендуются для разработчиков, поскольку дают несомненные преимущества от использования Win32 API в сравнении с потоковым вводом/выводом:

- высокая скорость дискового обмена данными;
- произвольная длина файла, точнее обрабатываются файлы с длиной до 2^{64} байт;
- обеспечивается переносимость программного кода на другие платформы (PowerPC, Alpha и MIPS).

Кроме того, Win32 API функции позволяют работать со следующими Windows объектами: 1) дисковые файлы; 2) каналы (pipes); 3) почтовые слоты (mailslots); 4) коммуникационные устройства (модемы); 5) консоли (consoles).

Второй особенностью Win32 функций при работе с файлами является возможность организации асинхронных операций чтения-записи. Это означает, что запустить, к примеру, процесс чтения информации можно в одном обработчике, а контролировать его окончание – в другом.

1. Стартовой функцией является **`CreateFile()`**, которая позволяет создать или открыть перечисленные выше объекты.

```
HANDLE CreateFile (
       LPCTSTR szFileName,
       DWORD   dwAccess,
       DWORD   dwShareMode,
       SECURITY_ATTRIBUTES* pSecurityAttr,
       DWORD   dwCreation,
       DWORD   dwFlags,
       HANDLE  hTemplateFile)
```

Здесь:

`szFileName`	имя Windows объекта. Имя может содержать пробелы;
`dwAccess`	режим открытия;
`dwShareMode`	режим доступа;
`pSecurityAttr`	указатель на структуру SECURITY_ATTRIBUTES. Обычно равен NULL;
`dwCreation`	режим создания;
`dwFlags`	флаги атрибутов. Особо следует упомянуть о флаге **`FILE_FLAG_OVERLAPPED`**, который задает асинхронные операции с файлом;

hTemplateFile дескриптор ранее открытого файла, атрибуты которого берутся за образец при создании нового файла.

Функция возвращает дескриптор файла или мнемоническую константу **INVALID_HANDLE_VALUE** в случае ошибки.

Приведем типичные примеры.

Открыть существующий файл на чтение можно следующим образом:

```
HANDLE hFile = CreateFile ("Just File.txt",
       GENERIC_READ,
       FILE_SHARE_READ, // разделение на чтение
       NULL,
       OPEN_EXISTING,
       FILE_ATTRIBUTE_NORMAL,
       NULL);
```

Создать новый или перезаписать существующий файл в монопольном режиме можно так:

```
HANDLE hFile = CreateFile ("New_Just_File.txt",
       GENERIC_WRITE,
       0,              // монопольный режим
       NULL,
       CREATE_ALWAYS, // перезапись существующего
       FILE_ATTRIBUTE_NORMAL,
       NULL);
```

2. После окончания работы с объектом, открытым с помощью функции **CreateFile()**, его *обязательно* нужно закрыть для освобождения внутренних ресурсов

```
CloseHandle(hFile);
```

3. Получить размер файла **hFile** произвольной длины можно через функцию

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

которая возвращает младшие четыре байта длины, а параметр **pSizeHigh** указывает на переменную, содержащую старшие четыре байта. Таким образом, результирующее значение длины файла составляет восемь байт.

4. Прочитать **nBytes** байт в буфер **pBuffer** можно через вызов:

```
BOOL ReadFile (HANDLE hFile,
       LPVOID  pBuffer,
       DWORD   nBytes,
       LPDWORD pBytesReaded,
       OVERLAPPED* pOverlapped)
```

при этом в переменную, адрес которой передается через **pBytesReaded**, будет записано реальное количество прочитанных байт. Если указатель **pOverlapped** равен NULL, то выполняется синхронное чтение информации. Это означает, что возврат из функции произойдет *только* после того, как будет выполнено чтение указанных **nBytes** байт или возникнет ошибка. Если параметр **pOverlapped** указывает на структуру **OVERLAPPED**, выполняется асинхронное чтение, когда **ReadFile()** немедленно возвращает управление, а для установления момента окончания чтения нужно предусмотреть специальный код.

5. Записать **nBytes** из буфера **pBuffer** в файл можно через вызов:

```
BOOL WriteFile(HANDLE hFile,
       LPCVOID pBuffer,
       DWORD   nBytes,
       LPDWORD pBytesWritten,
       OVERLAPPED* pOverlapped)
```

Функция возвращает **TRUE**, если операция выполнена успешно. Остальные параметры соответствуют функции **ReadFile()**.

6. Для перемещения файлового указателя используется вызов:

```
DWORD SetFilePointer (HANDLE hFile,
       LONG  Len,
       PLONG pLenHigh,
       DWORD dwMethod)
```

Новое положение файлового указателя задается двумя переменными типа **LONG** – **Len** (младшее двойное слово) и **pLenHigh** (старшее двойное слово). Параметр **dwMethod** задает направление перемещения и принимает три значения: **FILE_BEGIN**, **FILE_CURRENT** и **FILE_END**.

Функция возвращает младшее двойное слово нового положения файлового указателя, а старшее двойное слово передается через переменную, адрес которой указан на третьем позиции.

3.2. Задание. Создать приложение, выводящее в свое окно одну кнопку со стандартным видом и поведением и несколько кнопок с нестандартным видом и поведением. Все кнопки – класса “button”, стиль – BS_PUSHBUTTON. Нестандартные кнопки должны отличаться по своему внешнему виду: они должны менять свой вид в нажатом и отпущенном состоянии и при нажатии и отпускании выдавать звуковой сигнал. При реализации нестандартных кнопок следует воспользоваться созданием подклассов стандартной кнопки при помощи замены стандартной функции окна кнопки на новую.

Пояснения и указания.

Элемент управления кнопка. Рассмотрим особенности каждого стандартного элемента управления в отдельности. Материал по каждому элементу состоит из трех частей:

- стили, характерные для элемента управления;
- сообщения, поступающие в оконную функцию родительского окна;
- управляющие сообщения от родительского окна.

Стили кнопок. При создании кнопок функцией **CreateWindow()** можно использовать следующие дополнительные стили:

BS_PUSHBUTTON	Обычная кнопка
BS_DEFPUSHBUTTON	Создается так называемая “кнопка по умолчанию”. Этот стиль применяется только для диалоговых панелей. Тогда при нажатии клавиши Enter родительскому окну (диалогу) посылается сообщение о нажатии кнопки, имеющей этот стиль. Только одна кнопка на диалоговой панели может иметь такой стиль.
BS_NOTIFY	Если кнопка имеет данный стиль, родительскому окну посылаются сообщения **BN_DBLCLK**, **BN_KILLFOCUS** и **BN_SETFOCUS**. Заметим, что основное “кнопочное” сообщение **BN_CLICKED** посылается вне зависимости от этого стиля.
BS_OWNERDRAW	Кнопка с расширенными возможностями отрисовки, когда графическое отображение элемента становится обязанностью приложения, а не Windows. Кнопка с этим стилем посылает родительскому окно сообщения **WM_MEASUREITEM**, при создании кнопки, и **WM_DRAWITEM**, всякий раз, когда требуется перерисовка. Этот стиль несовместим с другими стилями.
BS_MULTILINE	На поверхности кнопки отображается многострочная надпись, а не однострочная – как в большинстве случаев.

BS_CENTER Центрирование текста надписи. Несовместимо со стилем
BS_VCENTER **BS_MULTILINE.**

Сообщения от кнопок, получаемые родительским окном. Ниже указаны сообщения, которые кнопка посылает своему родительскому окну. Часть из них является самостоятельными Windows сообщениями, а другие приходят в составе сообщения **WM_COMMAND.** В последнем случае старшее слово параметра **wParam** соответствует указанному нотификационному значению:

BN_CLICKED (WM_COMMAND) Кнопка нажата.

BN_DBLCLK (WM_COMMAND) Двойной щелчок левой клавишей мыши на поверхности кнопки. Это сообщение генерируется только для кнопок, имеющих стиль **BS_OWNERDRAW.**

BN_SETFOCUS BN_KILLFOCUS, (WM_COMMAND) Кнопка получает или теряет фокус ввода.

WM_CTLCOLORBTN Посылается родительскому окну, когда кнопке требуется перерисовать свой фон. Родительское окно должно вернуть дескриптор кисти для закрашивания фона.

WM_DRAWITEM Сообщение генерируется только для кнопок, имеющих стиль **BS_OWNERDRAW**. Уведомляет о том, что родительское окно должно выполнить графическое отображение кнопки. Параметры сообщения:
wParam - идентификатор кнопки
lParam - указатель на структуру **DRAWITEMSTRUCT.**
Описание полей структуры смотри в пункте 8.2.9.

Сообщения от родительского окна к кнопке. Родительское окно может изменить состояние кнопки, посылая ей сообщение **BM_SETSTATE**. Если параметр **wParam** установлен в значение **TRUE**, кнопка переводится с состояние “нажато”, если **wParam** равен **FALSE**, то в состояние “отжато”. Параметр **lParam** для данного сообщения не используется и должен быть равен нулю.

Кнопки-переключатели. Обычно переключатель не выделяют в самостоятельный элемент управления, а рассматривают его как разновидность кнопки, т.е. элемент управления, требующий нажатия.

При создании переключателей функцией **CreateWindow()** можно использовать следующие дополнительные стили:

BS_AUTOCHECKBOX Переключатель, который может находиться только в двух состояниях: включено и выключено.

BS_AUTO3STATE Переключатель, который может находиться в трех состояниях: включено, выключено и неактивно.

BS_AUTORADIOBUTTON Группа переключателей, при этом только один из группы может быть в состоянии включено, все остальные - выключены.

Сообщение от переключателей. Родительское окно получает от переключателей ранее рассмотренное сообщение **BN_CLICKED** в составе **WM_COMMAND.**

Сообщение от родительского окна к переключателям.

1. Родительское окно с дескриптором **hWnd** может запросить состояние переключателя, который имеет идентификатор **nID**, через сообщение **BM_GETCHECK**. Например:

```
int k=SendDlgItemMessage(hWnd,nID,BM_GETCHECK,0,0);
```

Возвращается одно из следующих состояний переключателя:

BST_CHECKED включен;
BST_UNCHECKED выключен;
BST_INDETERMINATE неопределенное состояние.

Последний код ответа может иметь только переключатель, для которого был определен стиль **BS_AUTO3STATE.**

2. Родительское окно может изменить состояние переключателя, посылая ему сообщение **BM_SETCHECK**, значение **wParam** параметра которого должно принимать одну из указанных выше величин, а параметр **lParam** не используется и должен быть равен нулю.

Структура DRAWITEMSTRUCT.

Структура **DRAWITEMSTRUCT** используется при отрисовке пунктов меню, а также для некоторых элементов управления (см. поле **CtlType** в приведенной ниже таблице), которые имеют стиль **OWNERDRAW**. Как уже было сказано, этот стиль сообщает Windows, что отрисовку необходимых элементов приложение берет под свой контроль. Обязанностью Windows остается заполнение полей структуры, которая содержит информацию, что и как нужно перерисовать. После этого указатель на структуру передается в функцию родительского окна для обработки. Заметим, что оконная функция будет получать *столько* сообщений **WM_DRAWITEM**, *сколько пунктов* требуется перерисовать. Перечислим допустимые значения для полей структуры:

CtlType Определяет тип элемента управления, который имеет стиль **OWNERDRAW:**
- **ODT_MENU** меню;
- **ODT_BUTTON** кнопка;
- **ODT_LISTBOX** обычный список;
- **ODT_LISTVIEW** расширенный список (Win32);
- **ODT_COMBOBOX** комбинированный список;
- **ODT_STATIC** статический текст;
- **ODT_TAB** набор закладок (Win32).

CtlID идентификатор элемента управления.
Не используется для меню.

itemID номер пункта меню или элемента списков, которые требуют перерисовки.

itemAction Определяет тип перерисовки:
- **ODA_DRAWENTIRE** перерисовка всего пункта;
- **ODA_FOCUS** пункт имеет фокус ввода;
- **ODA_SELECT** пункт является выбранным.

itemState Состояние пункта:
- **ODS_DEFAULT** обычное состояние;
- **ODS_FOCUS** имеет фокус ввода;
- **ODS_GRAYED** недоступен (только для меню);
- **ODS_DISABLED** недоступен (только для меню);
- **ODS_CHECKED** отмечен (только для меню);
- **ODS_SELECTED** выбран (только для меню).

hwndItem Дескриптор элемента управления.
Не используется для меню.

hDC Дескриптор контекста отображения.

rcItem Структура **RECT**, ограничивающая прямоугольную область, которая требует перерисовки.

itemData Двойное слово, ассоциированное с данным пунктом.

3.3. Задание. Написать двухпотоковую программу – тренажер работы на клавиатуре. Через определенный интервал времени (постепенно уменьшающийся) программа выводит в окно случайный символ. Это должен делать рабочий поток. Пользователь должен нажимать соответствующие клавиши клавиатуры, стараясь успеть за выводом символов на экран. Интерфейс с пользователем должен реализовываться в главном потоке. Для хранения информации о выведенных символах использовать такую структуру данных как очередь.

Пояснения и указания.

В операционных системах на базе Win32 организована *вытесняющая многозадачность* (multitasking) – это способность операционной системы выполнять несколько программ одновременно. В основе этого принципа лежит использование операционной системой аппаратного таймера для выделения квантов времени для каждого из одновременно выполняемых процессов. Если эти отрезки времени достаточно малы, и машина не перегружена слишком большим числом программ, то пользователю кажется, что все эти программы выполняются параллельно.

Многозадачность, реализованная в Win16, не была вытесняющей. Она использовала системный таймер для периодического прерывания выполнения одной задачи и запуска другой. Как это было реализовано? Переключение между задачами происходило только в тот момент, когда одна программа завершала обработку сообщения и возвращала управление Windows. Такую невытесняющую многозадачность называют также *кооперативной многозадачностью* потому, что она требует некоторого согласования между приложениями – одна программа могла парализовать работу всей системы, если ей требовалось много времени для обработки сообщения.

32-х разрядные версии Windows кроме многозадачности поддерживают еще и *многопоточность* (multithreading). Многопоточность – это возможность программы самой быть многозадачной. Программа может быть разделена на отдельные потоки выполнения (threads), которые, выполняются параллельно.

Итак, многозадачная операционная система обеспечивает одновременное исполнение двух или более приложений за счет выделения каждой из них процессорного времени. Многопотоковость же подразумевает использование нескольких параллельных потоков вычислений, относящихся к одной прикладной программе.

Таким образом, истинная Win32 программа – это совокупность одного процесса (process) и нескольких потоков (threads). *Процесс* – это исполняемый модуль, которому Windows выделяет память и другие системные ресурсы. *Поток* – это последовательность исполняемых команд. Процесс может состоять из единственного потока, а может содержать их несколько. И на процессы, и на потоки распространяется вытесняющая многозадачность. В результате отпадает необходимость встраивания в каждую прикладную программу механизма уступки управления, как это было в Win16, обеспечивающего выделение процессорного времени другим программам. Вместо этого, в операционных системах Win32, реализован сложный механизм диспетчеризации, обеспечивающий принудительное прерывание, или, другими словами, вытеснение активного потока в тот момент, когда наступает очередь другого. Это приводит к более четкой обработке многозадачности: при одновременном исполнении нескольких программ процессорное время распределяется между ними более последовательно и улучшается отклик программ на ввод. Чувствительность реакции программы является ее важным показателем, поскольку первое, что подме-

чает пользователь при неверной работе одной из программ, – это вялая реакция на манипуляции с мышью или клавиатурой.

Заметим, что распределяя процессорное время среди нескольких потоков, операционные системы Windows 95/98 или OS/2 создают иллюзию параллельно протекающих потоков. Только Windows NT при наличии нескольких процессоров действительно обрабатывает эти потоки одновременно, выделяя на обслуживание каждого из них свой собственный процессор. Подобное явление именуется симметричной мультипроцессорной обработкой (SMP).

В Win32 реализация фоновых действий, например, печать документа, не представляет особых трудностей. Создается рабочий поток, единственное назначение которого – это выполнение этих самых фоновых действий. При работе потока операционная система переключается с потока обработки ввода пользователя на фоновый поток и обратно. Более того, такая обработка происходит с максимальной эффективностью, поскольку когда поток для обслуживания ввода находится в ожидании, львиная доля процессорного времени выделяется фоновому потоку. Но, как только пользователь нажимает кнопку на клавиатуре или делает щелчок мышью, обработка фонового потока приостанавливается, и управление передается потоку ввода. Программа реагирует немедленно на все возникающие входящие сообщения.

Диспетчеризация потоков

Основным фактором, учитываемым при диспетчеризации, служит *приоритет выполнения* каждого потока. В Win32 каждому потоку присваивается свой приоритет - число от 0 до 31; чем больше число, тем выше приоритет. Приоритет 31 резервируется под особенно критичные операции, например, прием данных в реальном режиме времени. Приоритет 0 назначается операционной системой некоторым второстепенным задачам, выполнение которых происходит в то время, когда нет других задач. Большинство потоков работают с приоритетом в диапазоне от 7 до 11. Каждые несколько миллисекунд диспетчер операционной системы просматривает все работающие в системе потоки и передает управление в соответствии со следующими правилами:

1) процессорное время выделяется потоку с наивысшим приоритетом;

2) если потоков с одинаковым приоритетом несколько, то управление передается тому, который простаивает дольше других;

3) потоки с низким приоритетом никогда не вытесняют потоки с более высоким.

Здесь может возникнуть вопрос. Если в системе присутствует поток с приоритетом 10 и с приоритетом 9, то до выполнения потока с приоритетом 9 дело никогда не дойдет? Нет, это не так. Не забывайте, что Windows это событийная операционная система. Когда очередь сообщений потока с приоритетом 10 пуста, он переводится в состояние ожидания, до получения новых сообщений. При этом обрабатываются потоки с более низким приоритетом, а приостановленный поток не получает процессорное время до момента его активизации в ответ на какое-либо событие. Нужно сказать, что большинство потоков проводит длительное время как раз в состоянии ожидания ввода новых сообщений. В результате почти во всех случаях, кроме экстремальных, даже потоки с приоритетом 0 получают достаточно времени.

Отсюда вытекает вывод: не стоит стремиться поднять приоритет потоков собственного приложения с целью их более быстрой работы. Это, наверняка, приведет к обратному эффекту.

Проблемы многопоточной технологии

Архитектура многопоточного приложения обычно включает главный поток программы, который создается автоматически при запуске исполняемого модуля. Он создает

все окна и соответствующие им оконные процедуры, а также обрабатывает все сообщения для этих окон. Следовательно, главный поток должен иметь цикл обработки сообщений.

Все остальные потоки являются рабочими и служат для решения некоторых фоновых задач. Они не имеют оконных процедур и, значит, не обрабатывают сообщения операционной системы. Таким образом, рабочие потоки не выполняют задач, связанных с пользовательским интерфейсом.

Все потоки являются частями одного процесса, поэтому они разделяют все его ресурсы, такие как память, открытые файлы и прочее. Поскольку потоки разделяют память, отведенную программе, то они разделяют и статические переменные приложения. Однако каждый поток имеет свой собственный стек, т.е. автоматические переменные являются уникальными для каждого потока. Каждый поток также имеет свое собственное состояние процессора, которое сохраняется и восстанавливается при переключении между потоками.

Технология программирования для развитого многопоточного приложения – самая сложная задача, с которой приходится сталкиваться Windows программисту. Ее суть состоит в том, что в системе с вытесняющей многозадачностью поток может быть прерван в любой момент для переключения на другой поток, следовательно, может произойти неконтролируемое приложением взаимодействие между двумя потоками, которое будет нежелательным.

Создание рабочего потока

В первую очередь нужно написать так называемую “функцию рабочего потока”, тело которой и содержит операции, выполняемые рабочим потоком. Прототип функции одинаков для всех потоков

```
DWORD WINAPI ThreadFunc (LPVOID pData);
```

Имя функции рабочего потока не имеет значения, поскольку, как и для оконной процедуры, Windows использует ее адрес, а не имя. Все функции рабочего потока имеют только один параметр типа **LPVOID**. Следовательно, чтобы передать в функцию несколько параметров из вызывающего процесса, нужно определить собственную структуру, заполнить поля и передать ее адрес через **pData.**

Особенность функции рабочего потока состоит в том, что при выходе из нее *поток автоматически завершается.*

Используйте именно этот путь для окончания рабочего потока, а не принудительное завершение через специальные функции Win32 API. Это может быть источником трудноуловимых ошибок межпотокового взаимодействия.

Непосредственное создание рабочего потока происходит в момент вызова Win32 API функции **CreateThread()**, которая имеет прототип:

```
HANDLE CreateThread(
       SECURITY_ATTRIBUTES*  pThreadAttr,
       DWORD                 dwStackSize,
       THREAD_START_ROUTINE* pThreadFunc,
       LPVOID                pData,
       DWORD                 dwFlags,
       LPDWORD               pIdThread);
```

Первый параметр обычно не используется и равен **NULL**, второй – определяет размер стека для рабочего потока. Если он равен 0, Windows сама определяет необходимый размер. Третий параметр – это указатель на функцию рабочего потока, а четвертый – указатель блока памяти, который будет передан в **ThreadFunc()**. Флаг активизации рабочего потока обычно равен 0, что означает немедленный запуск потока. Через последний указатель буден возвращен идентификатор созданного потока. Однако больший интерес представляет дескриптор потока, возвращаемый функцией **CreateThread()**. Он позволяет вызывающему процессу проводить некоторые манипуляции с запущенными потоками.

Ниже представлен фрагмент кода, выполняющий запуск рабочего потока:

```
HANDLE       hThread;
DWORD        idThread;
THREAD_PARAM params;
// ....
hThread = ::CreateThread(NULL, 0, ThreadFunc,
                         &params, 0, &idThread);
if (!hThread) {
  // ошибки создания потока
}
```

Здесь **THREAD_PARAM** – некоторая структура, которая определяется в приложении. Она служит для передачи набора параметров в функцию потока.

Этот фрагмент содержит потенциальную ошибку, суть которой объяснена ниже.

При выходе из блока, запустившего поток, происходит разрушение локальных переменных **params** и **hThread**. Но уже запущенный поток не может "знать" об этом, он продолжает пользоваться данными из блока памяти, которого уже нет! Последствия этого – непредсказуемы.

Дескриптор **hThread** необходим для корректного освобождения ресурсов потока после его завершения следующим образом:

```
CloseHandle(hThread);
```

Выходом из сложившейся ситуации является объявление параметров **params** и **hThread** как глобальные или статические переменные.

Организация взаимодействия потоков

Как уже отмечалось, главная трудность технологии многопотокового программирования – это организация взаимодействия потоков. Приложение должно иметь возможность дуплексного обмена информацией. Это означает, что данные должны передаваться как от рабочего потока к процессу или, другими словами, в главный поток, так и в обратном направлении – от процесса в рабочий поток.

Рабочий поток – процесс

С первым направлением взаимодействия особых трудностей не возникает, поскольку здесь работает посылка сообщений. Напомним, что главный поток имеет функцию окна и, следовательно, может принимать сообщения от рабочего потока. Но какие сообщения? В заголовочном файле **windows.h** определена константа **WM_USER**, которая является верхней границей сообщений, используемых Windows для служебных целей. Это означает, что все истинные Windows сообщения имеют номера, меньшие чем **WM_USER**. Следовательно, для своих внутренних целей приложение безопасно может использовать сообщения со значением **WM_USER+n**, где **n>0**. При этом операционной системой гарантируется, что это не приведет к нежелательному эффекту коллизии сообщений, т.е. наложению одного на другое.

Как отправлять сообщения – синхронно или асинхронно? Ответ единственный. Следует использовать *только асинхронную* передачу через функцию **PostMessage()**, подразумевая, что в любой момент работа потока может быть прервана.

Процесс – рабочий поток

Рассмотрим типичные ситуации.

Рабочий поток должен начать обработку некоторых данных только после того, как эти данные будут подготовлены другим потоком. Как уведомить рабочий поток о том, что информация уже готова к обработке?

В силу каких-либо причин, рабочий поток должен немедленно завершить свою работу. Как сообщить ему об этом? Конечно, можно использовать принудительное снятие потока, но это, как правило, приводит к нежелательным побочным эффектам.

Подобные примеры можно продолжать, но уже ясно, что рабочий поток не может принимать сообщения, т.к. он не имеет функции окна.

1. Для координации действий потоков в такого рода ситуациях в Win32 предусмотрены специальные *формы синхронизации*. Одной из таких форм являются события (events). Объект событие может находиться только в двух состояниях: установлено и сброшено. Сам по себе объект событие не представляет интереса, он используется только в комбинации со специальными Win32 *функциями ожидания*.

Дескриптор на объект событие возвращает функция **`CreateEvent()`**:

```
HANDLE CreateEvent(
       SECURITY_ATTRIBUTES* pEventAttr,
       BOOL bManualReset,
       BOOL bInitialState,
       LPCTSTR lpName); //только для разных процессов
```

Здесь: указатель **`pEventAttr`**, как правило, равен NULL; переменная **`bManualReset`** определяет режим управления событием – вручную (**`TRUE`**) или автоматически (**`FALSE`**); переменная **`bInitialState`** задает стартовое состояние объекта – установлено (**`TRUE`**) или сброшено (**`FALSE`**).

Событие может быть именованным, тогда параметр **`lpName`** должен указывать на константную строку. Это имя используется в случае, когда требуется синхронизировать два или более процессов. Для взаимодействия процесс – рабочий поток данное имя не требуется, поэтому можно использовать **`NULL`** указатель.

Напомним, что все созданные или открытые Windows объекты требуют закрытия функцией:

```
CloseHandle((HANDLE)hObject);
```

когда отпадает необходимость в их использовании.

2. Второй трудностью межпотокового взаимодействия является доступ к разделяемым данным. Допустим, один поток модифицирует какие-то данные, а в это же время второй поток их читает. Если первый поток не завершил свои изменения, то второй получит неверные значения. Наилучший способ предотвратить такую ситуацию – это использование критических секций (critical sections), которые представляют собой разделы кода, во время выполнения которого текущий поток не может быть прерван, а все остальные переводятся в состояние ожидания.

Критическая секция должна быть проинициализирова, после чего функция

```
EnterCriticalSection(&cs);
```

начинает критическую секцию **`cs`**, а функция

```
LeaveCriticalSection(&cs);
```

завершает ее. При этом Windows гарантирует, что код программы, заключенный между этими двумя вызовами будет выполняться как бы в монопольном режиме, т.е. тогда, когда остальные потоки данного приложения будут «заморожены».

3. Следует упомянуть о третьей трудности, которая связана с динамическим захватом памяти в рабочем потоке. Эта проблема проявляется тогда, когда, возникает необходимость иметь постоянную область памяти, *уникальную для каждого потока*. Win32 API предоставляет ряд функций, поддерживающих такую специфическую память, которая называется локальной памятью потока (thread local storage или TLS). Однако лучшим решением следует признать пересмотр алгоритма решения задачи, и полный отказ от такого захвата в потоке.

Общая схема взаимодействия потоков

Рассмотрим общую схему взаимодействия главного и рабочего потоков. Определим структуру **SWT**, которая будет передаваться в функцию рабочего потока как параметр:

```
struct SWT {
       HANDLE hevBeg;  // событие начала
       HANDLE hevEnd;  // событие завершения
       CRITICAL_SECTION cs;
       // дополнительные поля
};
```

Структура, помимо полей, определяемых решаемой задачей, содержит два дескриптора событий и критическую секцию.

В главном потоке объявляются две статические переменные **swt** и **hThread**, а также проводится инициализация критической секции и создание объектов событий для управления рабочим потоком. Оба события используют "ручное" изменение состояния и первоначально установлены в "сброшено".

```
static SWT    swt;
static HANDLE hThread;

swt.hevBeg = CreateEvent(NULL, TRUE, FALSE, NULL);
swt.hevEnd = CreateEvent(NULL, TRUE, FALSE, NULL);
InitializeCriticalSection(&swt.cs);
```

Затем запускается рабочий поток с обслуживающей функцией **ThreadFunc()**, которой в качестве параметра передается указатель на структуру **swt**.

```
hThread                          =                          CreateThread(NULL,0,
                    ThreadFunc,&swt,0,&idThread);
```

После этого, возможно в другом обработчике функции главного окна приложения, устанавливается событие **hevBeg**, что является сигналом рабочему потоку к началу исполнения своего кода.

```
SetEvent(swt.hevBeg);  // Событие установлено
```

Теперь рассмотрим функцию рабочего потока

```
DWORD WINAPI ThreadFunc (LPVOID pData)
{
  // преобразуем указатель
  SWT* pswt = (SWT*)pData;
  // бесконечно ожидаем наступления события
/*1*/   WaitForSingleObject(pswt->hevBeg, INFINITE);
  // продолжаем цикл пока не установлено
  // события завершения рабочего потока
/*2*/   while (WaitForSingleObject(pswt->hevEnd, 0)
                                    != WAIT_OBJECT_0) {
       // вводим критическую секцию и ждем когда
       // остальные потоки будут заблокированы
       EnterCriticalSection(&pswt->cs);

       // некие действия
       // в "монопольном" режиме

       // сбрасываем критическую секцию
       LeaveCriticalSection(&pswt->cs);
  }
  // сбрасываем событие
  ResetEvent(pswt->hEvEnd);
  // выход из функции автоматически
  // завершает рабочий поток
  return 0;
}
```

В цикле, помеченном через /*1*/, функция рабочего потока будет находиться до тех пор, пока главный поток не установит событие **hevBeg**. Цикл, помеченный /*2*/, организован иным образом. Функция ожидания только проверяет состояние события **hevEnd**.

Если оно не установлено, то выполняется тело цикла, в противном случае – цикл заканчивается, что ведет к выходу из функции рабочего потока и завершению самого потока.

Вернемся в главный поток. Для того, чтобы завершить рабочий поток устанавливается событие **hevEnd**, после чего процесс ожидает момента его сброса в рабочем потоке.

```
SetEvent(swt.hevEnd);
while (WaitForSingleObject(swt.hevEnd, 200) ==
                                    WAIT_OBJECT_0);
// в данный момент рабочий поток завершен
```

Затем выполняется закрытие объектов

```
CloseHandle(hThread);
CloseHandle(swt.hevBeg);
CloseHandle(swt.hevEnd);
DeleteCriticalSection(&swt.cs);
```

Используете функцию ожидания **Sleep()**, только для рабочих потоков. При ее применении в главном потоке он блокируется и не обрабатывает поступающие сообщения.

3.4. Задание. Класс DIB для работы с Device Independent Bitmap. Изучить возможности отображения 2-х мерных изображений DIB формата. Разработать собственный класс DIB, обеспечивающий полнофункциональную работу с BMP изображениями (MFC класс CBitmap не дает таких возможностей). Минимальные требования: метод ReadDIBFile() для считывания изображения в BMP формате из файла, метод DrawDIB() для отрисовки BMP изображения.

Пояснения и указания.

Формат BMP (от слов BitMaP - битовая карта, или, говоря по-русски, битовый массив) является одной из форм представления растровой графики. Проще говоря, изображение представляется в виде матрицы прямоугольных точек, где каждая точка характеризуется тремя параметрами - x координатой, y координатой и цветом. Формат BMP разрабатывался изначально двумя корпорациями Intel и Microsoft, и в то время был одинаков для обеих операционных систем Intel OS/2 Warp и Microsoft Windows 2.x. Однако далее фирма Microsoft расширила формат, расширив структуры (при этом сохранив как обратную, так и прямую совместимость для несжатых разновидностей) и добавив поддержку компрессии. Добавилась поддержка сжатия без потерь PNG и RLE, а также сжатие с потерями JPEG. Казалось бы, JPEG и BMP, совместили несовместимое, однако это только на первый взгляд. На самом деле формат BMP - является родным не только для операционных систем Windows и OS/2, но и для различных аппаратных устройств (имеется ввиду его аппаратная версия DDB - будет описана далее). Родным в том смысле, что все операции графического ввода - вывода на экран (принтер и на некоторые другие устройства) в конечном итоге осуществляются посредством него (в том или ином его виде). Так вот, поскольку современные принтеры поддерживают прямой вывод изображений в форматах PNG и JPEG на устройство, и была введена их поддержка. Тем самым, обеспечив аппаратный вывод в рамках единого формата. Под вывод BitMaP-в, оптимизируется архитектура большинства видеоадаптеров. Для чтения и вывода в ОС Windows, предусмотрено много специальных функций и структур API (библиотека gdi32.dll и gdiplus.dll), которые помогают производить все необходимые операции на достаточно высоком логическом уровне. Delphi - еще более упрощает работу предоставляя нам класс надстройку над API - TBitMap, который здесь рассматриваться не будет поскольку хорошо описан во многих источниках. В заключение к разделу хочется развеять одну неопределенность. Windows поддерживает работу с тремя битмапоподобными форматами *.bmp, *.rle, *.dib. *.rle - это

сжатый битмап (как это следует из названия в формате RLE), полностью совместимый с bmp. *.dib - битмап версий Windows более чем 3.0. *.bmp - изначально предполагался быть совместимым с Windows 2.x, в последствии вероятно от этого отказались и сделали его мультиформатным. Данные форматы внутренне ни чем не друг от друга не отличаются (т.е. являются, по сути, псевдонимами *.bmp) и были введены для явного указания формата сжатия.

Аппаратно-зависимые и аппаратно-независимые битовые карты (Device-Dependenent and Device-Independent bitmap - DDB и DIB)

Аппаратно-зависимые или DDB битмапы используются windows для хранения изображений в памяти различных графических устройств (в частности в видеопамяти). Фактически такой битмап представляет собой урезанную версию аппаратно-независимого. Его данные формируются таким образом, чтобы соответствовать конкретному графическому режиму, кроме того, такой битмап содержит упрощенный заголовок. Например, для старого 16 цветного EGA/VGA видеоадаптера, такой битмап будет представлять собой 3 цветовые матрицы (для каждого из цветов), аппаратно-независимый битмап будет содержать всего одну матрицу разрядностью 4бита на пиксель.

Поскольку структура DDB битмапа меняется в зависимости от устройства к устройству, то он, как правило, создается прямо в памяти и не сохраняется в файл. Для сохранения в файл DDB конвертируется в DIB (т.е. аппаратно-независимый). В настоящее время графические ускорители оптимизируются под работу с DIB - универсальность в ущерб производительности (на самом деле эти потери незначительны). DDB битмап далее не будет описываться. Часто в литературе говорят битмап а подразумевают DIB (и наоборот) это не является грубой ошибкой.

Перед описанием структуры уточню, что структура битмапа в оперативной памяти повторяет файловую структуру, и все что верно для файла верно и для его образа в памяти, но неверно для DDB. Все структуры (записи) взяты из windows.pas с измененными именами, константы взяты из заголовочных файлов Borland C++ windows.h, wingdi.h.

Файл всегда состоит из трех частей.

1) Файловый заголовок - всегда структура TBitMapFileHeader - это единственная общая структура для всех типов и версии.

2) Затем для Windows версии 2.x и OS/2 идет структура (запись) TBitmapCoreInfo. Для всех остальных версий это TBitmapInfo

3) Массив данных - структура которого весьма разнообразна.

Программисты Windows - вероятно встречали только 'BM' (от слова BitMap, как вы уже, наверное, догадались). Не смотря на это, я полагаю, будет не лишним проверять, что за битмап нам передали (чтоб потом “не радовать” пользователя неожиданными ошибками при чтении правильных, с точки зрения, формата битмапов).

bfSize - это размер самого файла в байтах. В идеале все программы для того, чтобы убедиться, что перед ними действительно правильный bmp, должны, проверить, что bfType содержит "BM" (без кавычек), а, во-вторых, что bfSize равен действительному размеру файла. Хотя далеко не все программы используют это значение, оно должно быть верным, так как - это позволит нам убедиться в том, что файл был скопирован (или скачен) целиком.

bfReserved1 и bfReserved2 зарезервированы и должны быть нулями. Эти значение тоже желательно проверить, ведь в будущем они могут быть использованы для расширения формата. Естественно, что ваша программа не сможет их (такие файлы) прочитать, поэтому, узнав о ненулевых значениях можно правильно проинформировать пользователя, тем самым сэкономить его время.

bfOffBits - это один из самых важных полей в этой структуре. Он показывает, где начинается сам битовый массив относительно начала файла, который и описывает картинку. Несмотря на то, что это значение можно определить по концу таблицы цветов, рекомен-

дуется использовать именно это значение, для совместимости с возможными новыми вариациями формата.

Все структуры являются расширением первой TBitmapCoreInfo - поэтому в последующих структурах будут описаны только новые поля.

bcWidth и bcHeight - ширина и высота изображения в пикселях. В последующих структурах это будут уже двойные слова (из-за этого и нет полной совместимостей новых структур с TBitmapCoreHeader). Значение bcHeight (для версий Windows 3.x, 95, NT) может быть отрицательным. В этом случае модуль bcHeight определяет действительную высоту, а строки изображения читаются в обратном порядке, т.е. сверху вниз (обычно снизу вверх). Кроме того, такие изображения всегда несжатые (т.е. BI_RGB или BI_BITFIELDS).

bcPlanes - задает количество плоскостей или цветовых слоев (помните я упоминал что DDB - могут иметь 3 цветовые плоскости например R G B или C M Y- связанные с особенностями графического устройства). Сохраняемая версия битмапа т.е. DIB поддерживает пока одну общую для всех цветов плоскость, разрядность цвета этой плоскости задается bcBitCount. Попытки установить значения отличные от единицы вызывают ошибку при передаче структуры api функциям.

bcBitCount - определяет разрядность цвета. Допустимы следующие варианты

0 - изображение только PNG и BMP допустимо только версии Windows 98/Me, Windows 2000/XP

1 - монохромное изображение (поддерживается всеми версиями). Каждый пиксель представлен одним битом данных, т.е. один байт содержит информацию о цвете 8 последовательно идущих пикселей. Цвет первого пикселя определяется состоянием старшего бита первого байта (и так далее), если его значение равно единице, то цвет пикселя будет определяться первой записью таблицы цветов (считается от нуля). Вообще значение цвета определяется по RGB (для BI_RGB версии) составляющим, таблицы цветов, по индексу.

4 - 16-ти цветное изображение. Каждый пиксель представлен 4 битами (поддерживается всеми версиями).

8 - изображения с количеством цветов до 256. 1 байт - 1 пиксель (поддерживается всеми версиями)

16 - поддерживается не менее чем Windows 95, и является достаточно редким. Количество цветов зависит от версии Windows и выбранного формата сжатия и может быть как 2 в 16 либо 2 в 15 степени. Это самый запутанный вариант. Начнем с того, что он беспалитровый, то есть каждые два байта (одно слово WORD) в растре однозначно определяют один пиксель. Но вот что получается: битов-то 16, а компонентов цветов - 3 (Красный, Зеленый, Синий). А 16 никак на 3 делиться не хочет. Поэтому здесь есть две варианта. Первый - использовать не 16, а 15 битов, тогда на каждую компоненту цвета выходит по 5 бит. Таким образом, мы можем использовать максимум 2 в 15 = 32768 цветов, и получается тройка R-G-B = 5-5-5. Но тогда за зря теряется целый бит из 16. Но так уж случилось, что наши глаза среди всех цветов лучше воспринимают зеленый цвет, поэтому и решили этот один бит отдавать на зеленую компоненту, то есть тогда получается тройка R-G-B = 5-6-5, и теперь мы может использовать 2 в 16 = 65536 цветов. Но что самое неприятное, что используют оба варианта. В MSDN предлагают для того, чтобы различать, сколько же цветов используется, заполнять этим значением поле biClrUsed (будет описано ниже) из структуры BITMAPINFOHEADER. Чтобы выделить каждую компоненту надо использовать следующие маски. Для формата 5-5-5: 001Fh для синей компоненты, 03E0h для зеленой и 7C00h для красной. Для формата 5-6-5: 001Fh - синяя, 07E0h - зеленая и F800h красная компоненты соответственно.

24 - а это самый простой формат. Количество цветов 2 в 24 степени. Здесь 3 байта определяют 3 компоненты цвета. То есть по компоненте на байт. Просто читаем по структуре RGBTRIPLE(для Windows версии 2.x и OS/2) и используем его полягbtBlue, rgbtGreen, rgbtRed. Они идут именно в таком порядке (поддерживается всеми версиями).

32 - Здесь 4 байта определяют 3 компоненты, т.е. по-прежнему количество цветов 2 в 24 степени. Но, правда, один байт не используется. Его можно отдать, например, для альфа-канала (прозрачности).

3.5. Задание. Звездное небо. Обеспечить графический вывод изображения звездного неба, на котором некоторые звезды в случайном порядке “зажигаются” и “гаснут”. Для хранения информации о звездах использовать такую структуру данных как односвязанный список. Программа должна быть двухпотоковая, за отрисовку постоянно меняющегося изображения должен отвечать рабочий поток. Пользователь должен иметь возможность изменения количества звезд и их времени “жизни”.

Пояснения и указания.

См. пояснения к заданию 3.3.

3.6. Задание. Программа преобразования чисел. Написать программу перевода любого десятичного числа в двоичную и шестнадцатеричную системы счисления. При запуске программа отображается в системном “трее” (в правом нижнем углу, у часов). Пользователь помещает число в буфер обмена и делает двойной щелчок по иконке программы. В результате отображается диалоговая панель с результатом. Предусмотреть анализ нечисловых данных в буфере обмена.

Пояснения и указания.

Системный трей (system tray) - это небольшая область в панели задач в нижнем правом углу монитора, там где у вас расположены часы. Эту область можно настроть для отображения значков запущенных приложений, которые при работе создают возле часов маленькую иконку. Такими программами можно управлять прямо из трея.

У среднестатистического пользователя в трее располагается три - семь значков различных программ, которые открыл он или которые были запущены при старте системы. По умолчанию на только что установленной Windows в системной трее, обычно отображаются значки изменения раскладки клавиатуры, громкости, индикатора сети и часов.

Например, в системном трее компьютере можно расположить иконку программы, через которую можно осуществлять управление приложениями. При этом на рабочем столе и в командной строке иконок для запуска этого программного обеспечения нет, что очень удобно.

Свернуть в трей означает скрыть все проявления программы с экрана и отобразить в трее значок, ассоциированный с программой, вернее, с неким окном, причём необязательно к процессу принадлежащим. Для того чтобы свернуть в трей нужно: иметь дескриптор окна HWND, придумать идентификатор иконки и вызвать

BOOL Shell_NotifyIcon (DWORD dwMessage, PNOTIFYICONDATA lpdata);

Здесь:

dwMessage - параметр, а точнее команда, которая указывает этой функции, что именно она должна делать. Может принимать следующие значения констант:

NIM_ADD - добавляет иконку в трей;

NIM_DELETE - удаляет иконку из системного трея;

NIM_MODIFY - меняет (обновляет) иконку в системном трее.

lpData – указатель на структуру NOTIFYICONDATA, которая содержит всю информацию о добавляемой, удаляемой или изменяемой иконке. Вот основные поля данной структуры:

DWORD cbSize - размер в байтах данной структуры;

HWND Wnd - дескриптор того окна, которое будет получать сообщения от иконки;

UINT uID - идентификатор иконки;

UINT uFlags - содержит константы, означающие, какие поля используются в структуре;

NIF_ICON - hIcon содержит верную информацию;

NIF_MESSAGE - uCallbackMessage содержит верную информацию;

NIF_TIP -szTip содержит верную информацию;

UINT uCallbackMessage - назначенный вами идентификатор сообщения, которое будет получать приложение от иконки.

Список литературы

1. Боровской И.Г. Технология разработки программных систем: Учебное пособие / Боровской И. Г. — 2012. 260 с. https://edu.tusur.ru/lecturer/publications/2436
2. Специализированная подготовка разработчиков бизнес приложений : Учебное пособие / Боровской И. Г., Матолыгин А. А., Колесникова С. И. — 2012. 256 с. https://edu.tusur.ru/lecturer/publications/2532