Лабораторная работа №1. Знакомство со средой разработки Borland Delphi.

Цель работы: ознакомление с особенностями Borland Delphi как интегрированной среды разработки 32-разрядных приложений под Windows на диалекте языка Паскаль. Написание и отладка простейшей программы. Изучение работы со Справкой по Win32 API.

Borland Delphi

Borland Delphi 2.0 (и более поздних версий) является удобным инструментом для разработки 32-разрядныз приложений для Windows. В состав Delphi входят:

- компилятор с диалекта языка Паскаль Borland Pascal 8.0 with objects, позволяющий создавать 32-разрядные исполнимые файлы для Windows;
- интегрированная среда разработчика, включающая текстовый редактор, отладчик и средства для визуального проектирования форм;
- библиотека классов VCL (Visual Components Library), позволяющая создавать Windows-программы разработчикам, имеющим минимальные знания о логике работы Windows.

Так как в предлагаемом курсе лабораторных работ объектом изучения является именно то, как Windows общается с прикладными программами, библиотекой VCL, объектно-ориентированными расширениями Паскаля и средствами визуального проектирования форм **пользоваться запрещается**. Таким образом, Delphi используется исключительно как 32-разрядный компилятор Паскаля, снабженный отладчиком.

Для программиста, работавшего на Турбо-Паскале для DOS фирмы Borland, переход на Паскаль, встроенный в Delphi, особенно с языковой точки зрения, не составляет труда, особенно если не использовать объектно-ориентированные расширения. Гораздо больше проблем возникает с освоением новых принципов построения программ, обусловленных ОС Windows. К основным языковым отличиям следует все же отнести следующие:

- Тип integer означает 32-разрядное целое со знаком, а не 16-разрядное, как раньше. 16-разрядные целые имеют тип SmallInt.
- Все указатели — линейные 32-разрядные, они больше не состоят из 16-разрядных сегмента и смещения.
- Строковая константа может передаваться в качестве фактического параметра типа PChar, при этом строка завершается нулем.
- Функция может возвращать любое значение, в том числе типа record, а не только целое, вещественное, символьное, строковое и указатель.
- Счетчик цикла for после выхода из цикла считается неопределенным.

Справка по Win32 API

Программа, написанная для Windows, общается с операционной системой посредством вызова функций API, присутствующих в системных динамически связываемых библиотеках (DLL). Набор этих функций весьма обширен, справочная информация по ним в виде Help-файла Windows занимает десятки мегабайт. Дополнительной неприятной особенностью этой документации является то, что она ориентирована на язык C, т.е. заголовки функций с описанием параметров приводятся именно на этом языке. Поэтому данный курс лабораторных работ ориентирован на студентов, хорошо знакомых с языками C и Паскаль и способных самостоятельно транслировать прототип функции C к "паскалевскому" виду. К сожалению, имеется только версия Справки на английском языке.

Дополнительной помощью в такой трансляции является то, что в комплект поставки Delphi входят исходные тексты модулей (units) языка Паскаль, реализующих интерфейс с

системными DLL. Прежде всего, это модули WINDOWS.PAS и MESSAGES.PAS, расположенные в каталоге LabWin. Если при прочтении статьи Справки возникает неясность в вопросе, как описан интерфейс той или иной функции в Delphi, всегда есть возможность заглянуть в упомянутые исходные файлы и найти описание нужной функции средствами встроенного текстового редактора.

Основные разночтения возникают обычно, когда функция API имеет в качестве параметра указатель на тот или иной объект. В языке Паскаль это может быть реализовано двояко — объявлением соответствующего параметра функции параметром-переменной (var-параметром) или объявлением его действительно указателем.

"Венгерская нотация"

Имена формальных параметров функций, приводимых в Справке по Win32 API, записаны в так называемой "венгерской нотации". Первые буквы таких идентификаторов позволяют судить о типе формального параметра, а дальше как правило идет осмысленное английское название. Например:

fErase — признак (флаг) необходимости стирания содержимого окна при перерисовке;

hWnd — хэндл (описатель, внутренний номер) окна;

psString — указатель на некоторую строку;

iVScrollPos — позиция движка на вертикальной полосе прокрутки окна.

Ниже приводится таблица "расшифровки" типов параметров по первым буквам названия.

Префикс	Тип данных
c	символ (char)
by	байт, беззнаковое целое от 0 до 255 или символ (byte)
n	короткое целое (16-разрядное, -32768..32767)
i	целое (32-разрядное со знаком)
x, y	целое, используемое в качестве координат X и Y
cx, cy	целое, используемое в качестве длины или счетчика вдоль осей координат X и Y
b, f	логическое (boolean) или флаг
w	слово, беззнаковое короткое целое (16-разрядное, 0..65535)
L	длинное целое (32-разрядное со знаком)
dw	двойное слово (32-разрядное без знака)
fn	функция
s	строка
sz	строка, завершающаяся нулем
h	хэндл, описатель (handle)
p	указатель (pointer)

Простейшая программа

В качестве первой программы при изучении любого языка программирования обычно приводится нечто, просто выдающее на экран приветствие вроде "Hello, world!". Многие учебные руководства по Windows содержат утверждение, что с Windows такой номер не пройдет и даже в простейшей программе надо написать пару десятков строк. на самом деле это не совсем так, и это иллюстрирует программа HELLO.PAS, расположенная в каталоге LAB1. Она использует функцию Windows, как раз позволяющую вывести на экран окно с сообщением и даже несколькими стандартными кнопками (например, Ok, Cancel, Help и т.д.)

Конечно, утверждение про несколько десятков строк не лишено основания — но только в том случае, если в программе описываются СВОИ окна и их поведение; с другой стороны, окна описываются в подавляющем большинстве Windows-программ, и это считается само собой разумеющимся.

(Чтобы запустить программу, выберите в меню "Файл" пункт "Открыть проект" (не файл!), в появившемся диалоге укажите, что открывается Паскаль-проект, а не проект Delphi, и выберите соответствующий файл.)

Лабораторная работа №2

Исследование общих принципов реализации многозадачности

Работа проводится с использованием программной модели планировщика, реализующей:

- поддержку (псевдо)параллельного выполнения до 10 потоков программы;
- выполнение потоков в режиме вытесняющий и невытесняющей многозадачности;
- поддержку механизма критических секций (до 10 критических секций).

Модель планировщика реализована на языке Турбо Паскаль 7.0 для DOS и представляет собой модуль (unit) Турбо Паскаля, написанный с использованием вставок на языке ассемблера процессора Intel 8086/88.

> **Внимание!** На современных машинах (более мощных, чем PII-400MHz) при использовании в Паскаль-програме модуля CRT возникает ошибка 200 "Деление на ноль". Для устранения этой неприятности необходимо пользоваться исправленным модулем CRT или прилагаемой "заплаткой" **rdelay.tpu**. Данный модуль должен фигурировать во фразе **uses** основной программы непосредственно перед **crt**; кроме того, компиляция обязательно должна осуществляться на диск.

Теоретические положения

Термины и определения

Планировщиком называется часть операционной системы, ответственная за организацию (псевдо)параллельного выполнения нескольких задач и потоков в рамках одной задачи. Изучаемая модель планировщика управляет потоками.

Задачей (*task, application*) или **процессом** (*process*) называется выполняемая программа. В многозадачной ОС могут одновременно выполняться несколько задач в режиме разделения процессорного времени (или действительно одновременно, если процессоров несколько). Каждая задача имеет свой стек и, как правило, собственное адресное пространство; для задач планировщик в специальной записи состояния отслеживает выделенные ресурсы (открытые файлы, занятые блоки памяти и т.д.)

Потоками (*threads*) называются ветви (процедуры) в рамках одной задачи, выполняемые параллельно. Потоки разделяют ресурсы задачи, выполняются в одном адресном пространстве, но каждый поток имеет свой собственный стек.

Термин "**невытесняющая** (*non-preemptive*) **многозадачность**" означает, что переключение процессора с одной задачи на другую происходит, когда выполняемая в данный момент задача явно или неявно (в любом случае – через вызов каких-то системных функций) передает управление планировщику, чтобы тот мог выполнить переключение.

При **вытесняющей** (*preemptive*) **многозадачности** запуск процедуры переключения происходит независимо от "желания" текущей задачи, текущая задача принудительно прерывается в произвольный момент, и активизируется другая. Для инициации переключения задач обычно используется аппаратное прерывание от таймера.

Критической секцией (КС) называется участок кода некоторой программы А, выполняемой обычно в режиме вытесняющей многозадачности, в пределах которого запрещено выполнение другими параллельно исполняемыми программами некоторых действий, которые могут помешать правильной работе рассматриваемого участка кода программы А. В виде критических секций обычно оформляются последовательности команд, для которых

важно соблюсти неразрывность выполнения "одна-за-одной". При этом **вход в критическую секцию НЕ означает, что задача не может быть прервана и управление не может быть передано другой задаче!**

Критическая секция начинается командой входа в критическую секцию и заканчивается командой выхода. Планировщик анализирует запрос процесса на вход в критическую секцию и, если вход возможен, начинают выполняться команды критической секции. Если вход в критическую секцию невозможен, т.е. какой-либо другой процесс находится в состоянии выполнения критической секции, запрашивающий процесс переводится в состояние ожидания и выполняются другие процессы, пока не сложится ситуация, делающая для запрашивающего процесса возможным вход в критическую секцию.

Многие операционные системы, например - Windows 95, поддерживают нумерованные критические секции. С их помощью можно осуществить разделение доступа процессов к определенным ресурсам, не блокируя работу тех процессов, которые не используют ресурс. Для этого процессы должны при входе в КС указывать идентификатор (номер) КС. Если некоторый процесс выполняет КС с определенным номером, то другие процессы не могут войти в КС с тем же номером, но могут выполнять КС с другими номерами. Например, пусть для вывода на печать процесс должен войти в КС1, а для вывода в файл - в КС2. При этом, если один процесс печатает (выполняет команды, защищенные КС1), то ничто не мешает другому процессу работать с файлом (выполнять КС2), но начать печатать (войти в КС1) он сможет только тогда, когда первый процесс выйдет из КС1.

Команды входа и выхода из критической секции представляют собой так называемые "**критические скобки**".

Использование модели планировщика для организации параллельных вычислений

Изучаемая в данной работе модель планировщика реализует лишь часть функций планировщика реальной ОС. Модель написана для процессора Intel 8086/88 (или, соответственно, для реального режима работы более старших процессоров данного семейства). В связи с этим функции выделения памяти и поддержка выполнения нескольких задач не реализованы, реализовано лишь выполнение процедур DOS-программы в режиме разделения времени с использованием механизма невытесняющей или вытесняющей многозадачности и критических секций. В режиме вытесняющей многозадачности планировщик циклически выделяет всем активным процессам одинаковые кванты времени.

Планировщик реализован в качестве множества функций модуля H314TASK.TPU (исходный текст - файл H314TASK.PAS). Поэтому в разделе **uses** основной программы должен упоминаться этот модуль.

Каждый параллельный поток основной программы реализуется в виде процедуры без параметров, обязательно откомпилированной с дальней моделью вызова (опция компилятора $F+). Обычно тело процедуры представляет собой цикл, конечный или бесконечный. Рекомендуется отключить проверку переполнения стека (опция компилятора $S-), так как планировщик переключает стек в процессе своей работы. Процедуры ни в коем случае не должны завершаться, поэтому рекомендуется в конце каждой процедуры-потока ставить вызов процедуры LoopBack, зацикливающей поток (возврат из процедуры LoopBack никогда не производится).

Для запуска параллельных вычислений необходимо выполнить следующие действия:

1) Зарегистрировать потоки в таблице потоков планировщика при помощи функции

```
function RegisterThread(ThreadAddr:pointer; StackSize:word):integer;
```

Параметры функции - адрес точки входа в процедуру потока и размер стека потока. Функция возвращает хэндл (номер) потока, который идентифицирует его в системе. Например, можно написать:

```
ht1 := RegisterThread(@proc1,8000);
```

2) Запустить выполнение зарегистрированных потоков в режиме разделения времени, выполнив процедуру

```
procedure ExecuteRegisteredThreads(UseTimer:boolean);
```

Параметр UseTimer определяет, будет ли планировщик производить переключение задач по прерываниям таймера (TRUE, вытесняющая многозадачность) или для этого в теле каждой задачи должна периодически вызываться процедура SwitchThreads (FALSE, невытесняющая многозадачность).

3) Очистить информацию в таблице потоков планировщика и освободить память, занимаемую потоками под стек, с помощью процедуры ClearThreads.

При запуске процедуры ExecuteRegisteredThreads выполнение дальнейших команд основного потока программы приостанавливается до тех пор, пока все параллельные потоки не будут завершены вызовом процедуры StopThread с параметром – номером останавливаемого потока (возвращается процедурой RegisterThread). Процедура StopThread может вызываться из любого параллельного потока, а не только из того, который останавливается. Это позволяет логически выделить один из параллельных потоков в качестве "главного" (управляющего) и сосредоточить управление параллельными вычислениями в нем.

Для управления выполнением потоков предусмотрены следующие процедуры:

StopThread(num: word); - останавливает выполнение потока, запрещая передачу ему управления планировщиком. Передача управления может быть восстановлена процедурой RunThread. Когда все потоки остановлены, выполнение процедуры ExecuteRegisteredThreads завершается и продолжается выполнение вызвавшего ее основного потока программы.

RunThread(num:word); - возобновляет выполнение потока, ранее остановленного командой StopThread, включая его в число потоков, обслуживаемых планировщиком. Управление запускаемому потоку не передается немедленно, он активизируется "на общих основаниях" в соответствии с алгоритмом работы планировщика.

SwitchThreads; - передает управление планировщику для активизации им другого потока. Эта процедура в режиме невытесняющей многозадачности всегда должна периодически вызываться в теле потоков для обеспечения переключения задач. В режиме вытесняющей многозадачности вызов этой процедуры позволяет "досрочно" отдать управление другому потоку, например, если текущий поток в цикле ожидает какого-либо события, в противном случае ожидающий поток будет просто впустую расходовать время процессора.

LoopBack; - бесконечный цикл, из которого нет выхода. Эту процедуру рекомендуется вызывать в конце каждой процедуры потока.

EnterCritical(num: integer); - вход процесса в критическую секцию с указанным номером. Если вход возможен, поток продолжает выполняться дальше, блокируя вход в критическую секцию с данным номером для всех остальных потоков. Если вход невозможен, то управление передается другим потокам, и выполнение команд потока возобновится только тогда, когда блокирующий критическую секцию поток освободит ее.

LeaveCritical(num: integer); - выход процесса из критической секции с указанным номером. После выполнения этой команды другие процессы получают возможность войти в критическую секцию с указанным номером. Если на момент выхода из КС в системе имеются процессы, ожидающие входа в эту КС, то покидающий КС процесс досрочно прерывается и активизируется следующий процесс (не обязательно ожидающий входа в КС, просто - следующий).

Пример программы, использующей описываемый планировщик, можно найти в файле TASKSWAP.PAS.

Реализация планировщика

Поскольку наш планировщик не занимается выделением потокам ресурсов (за исключением стека), то основной его функцией является обеспечение переключения между выполняемыми параллельно потоками.

Состояние потока в каждый момент времени описывается состоянием регистров процессора. Прервав выполнение команд одного потока с помощью аппаратного или программного прерывания, можно затем организовать подмену значений в регистрах на значения для другого потока, после чего выход из процедуры обслуживания прерывания будет производиться уже в новый участок кода, соответствующий другому потоку.

Для поддержания информации о потоках планировщик использует 29-байтные записи (массив TS, записи типа TThreadStateWord), содержащие значения всех регистров процессора (в том числе флагов, сегментных регистров, счетчика команд и указателя стека), а также признак активности потока, указывающий, следует ли планировщику выделять время этому потоку. Информация о размерах стеков потоков содержится в массиве Stacks.

Процедура переключения потоков назначается в качестве обработчика прерывания 8, возникающего аппаратно по таймеру (процедура TimerHandler). Эта процедура извлекает состояние регистров, бывшее до вызова прерывания, из стека и сохраняет их в записи таблицы потоков (TS), соответствующей активному потоку. Затем определяется новый поток из числа активных, который подлежит активизации (перебор записей в таблице потоков производится циклически, что гарантирует, что каждый активный поток рано или поздно получит управление). Из записи таблицы потоков, соответствующей активизируемому потоку, извлекается состояние регистров (значения большинства регистров подменяются в стеке, так как будут извлекаться оттуда при возврате из процедуры прерывания, значение в паре SS:SP, адресующей стек, заменяется непосредственно). Возврат из обработчика прерывания производится уже в новый поток с восстановленными значениями его регистров.

> Некоторые сведения о работе компилятора:
>
> Наш обработчик скомпилирован с ключевым словом **interrupt**. Это означает, что точка входа и точка выхода будут оформлены компилятором следующим образом:
>
> Вход:
>
> (Перед этим при прерывании в стек были помещены точка возврата (значения в CS:IP) и регистр флагов, прерывания также автоматически запрещены.)

```
push   ax
push   bx
push   cx
push   dx
push   si
push   di
push   ds
push   es
push   bp
```

```
mov    bp,sp     ; сохранение SP в BP
mov    ax,@data  ; настройка DS на сегмент данных программы
mov    ds,ax
.............
```

Выход:

```
pop    bp
pop    es
pop    ds
pop    di
pop    si
pop    dx
pop    cx
pop    bx
pop    ax
iret  ; извлечение флагов и CS:IP
```

Для реализации "досрочного" переключения потоков и невытесняющей многозадачности реализована процедура SwitchThreads, которая просто генерирует прерывание 8 программным путем, при этом взводится флаг DisableHardwareEvents, блокирующий вызов системного обработчика прерывания 8, в котором производится сброс контроллера прерываний и обновление текущего времени: ведь прерывание вызвано не аппаратурой, а программой. Функции по переключению потоков при этом выполняются. Другой флаг, PreemptiveSwitch, позволяет блокировать переключение потоков по аппаратному прерыванию таймера, что дает возможность реализовать невытесняющую многозадачность.

Для поддержания информации о состоянии потоков используется массив записей, имеющих следующую структуру:

```
type TThreadStateWord=record
       BP_reg, ES_reg, DS_reg, DI_reg, SI_reg, DX_reg,
       CX_reg, BX_reg, AX_reg, IP_reg, CS_reg, Flags:  word;
       Stack:pointer;
       Active:boolean;
     end;
```

Порядок следования регистров в записи в точности повторяет порядок их следования в стеке при вызове процедуры обработки прерывания, что позволяет удобно манипулировать регистрами, Stack содержит текущее состояние пары SS:SP, Active показывает, активен процесс или остановлен.

При регистрации потока процедурой RegisterThread значения регистров в соответствующей записи обнуляется, в поле Active заносится TRUE, под стек потока в куче захватывается блок памяти указанного размера. В поле Stack заносится указатель на последнее слово в указанном блоке, так как при нарастании стека указатель стека SP уменьшается. Значение поля Stack будет выбираться в качестве начального состояния стека (SS:SP) соответствующего потока, в нем же будет сохраняться состояние стека потока при переключении потоков.

Действия процедуры ExecuteRegisteredThread состоят в том, что производится очистка таблиц критических секций, инициализируется обработчик прерывания 8, после чего запускается цикл ожидания. Условием выхода из цикла ожидания является значение TRUE в переменной ParallelFinished, свидетельствующее о том, что все зарегистрированные потоки остановлены. При выходе восстанавливается старое значение обработчика прерывания 8. Если происходит запуск параллельных вычислений в режиме невытесняющей многозадачности, то перед входом в цикл управление передается одному из потоков командой SwitchThreads, так как по аппаратным прерываниям таймера переключение в этом случае не производится.

Критические секции

Для реализации механизма критических секций служат процедуры EnterCritical и LeaveCritical. Разумеется, использование критических секций имеет смысл только при вытесняющей многозадачности, в случае невытесняющей многозадачности просто не следует отдавать управление планировщику в критических участках кода.

> В данном случае, когда программа написана для реального режима DOS, можно защитить критический участок кода от какого бы то ни было вмешательства, просто запретив прерывания командой CLI. Это так, но в реальных многозадачных операционных системах пользовательская программа как правило не имеет доступа к системе прерываний вообще, а команды типа CLI/STI вызывают исключительную ситуацию в процессоре, обрабатываются ОС и в конце концов к общему запрещению прерываний как правило не приводят, в лучшем случае – сигналы о прерываниях перестают поступать в приложение.

Внутри планировщика поддерживаются два вспомогательных массива: CriticalTable и CriticalTableWait. Индекс в массиве соответствует состоянию критической секции с соответствующим номером. Если ячейка в массиве CriticalTable содержит 0, то любой процесс способен сразу войти в критическую секцию с соответствующим номером выполнив процедуру EnterCritical. При этом значение в ячейке будет изменено на ненулевое.

Если значение в ячейке CriticalTable при выполнении EnterCritical не равно 0, то процесс блокируется перед входом в критическую секцию и ожидает ее освобождения, а для индикации наличия ожидающих процессов увеличивается счетчик в соответствующей массива CriticalTableWait. Так как процесс переходит в состояние ожидания, выполняется процедура SwitchThreads, активизирующая следующий процесс, независимо от того, разрешены ли аппаратные прерывания от таймера.

При выполнении процедуры LeaveCritical соответствующая в массиве CriticalTable сбрасывается в 0. Если при этом в соответствующей ячейке массива CriticalTableWait содержится ненулевое значение, свидетельствующее о наличии ожидающих процессов, то происходит передача управления следующему процессу с помощью SwitchThreads. В противном случае, если ни один процесс не ожидает входа в освобождаемую критическую секцию, текущий процесс продолжает свое выполнение.

Ситуация взаимной блокировки

При использовании механизмов блокировки процессов, к которым относятся критические секции, возможна ситуация, когда два или более процесса блокируют друг друга, т.е. не могут далее выполняться, так как для продолжения одного нужно, чтобы другой освободил некий ресурс, а другой процесс ожидает аналогичного действия от первого.

Простейшим примером такой потенциально опасной ситуации может служить следующая ситуация:

Процесс 1	**Процесс 2**
EnterCritical(1);	EnterCritical(2);
EnterCritical(2);	EnterCritical(1);
..... действия секции	..... действия секции .....
LeaveCritical(1);	LeaveCritical(1);
LeaveCritical(2);	LeaveCritical(2);

В этом случае возможен, например, вариант, когда после входа в критическую секцию 1 первый процесс прерывается, начинает исполняться второй процесс, там происходит вход в КС2 и остановка в ожидании освобождения КС1, занятой первым процессом. Когда же первый процесс получит управление, он продолжит свое выполнение и попытается войти в КС2, только что занятую вторым процессом, и также остановится в ожидании. В результате выполнение процессов 1 и 2 будет остановлено.

Очевидно, что ситуация может развиваться и иначе: если между входом в 1 и 2 КС первого процесса не произойдет прерывания, то блокировки не будет.

В изложенной ситуации ясен и путь решения проблемы: если захватывать и освобождать КС в обоих процессах в одинаковой последовательности, то взаимная блокировка будет невозможна.

К сожалению, в реальной жизни встречаются более запутанные ситуации со взаимной блокировкой нескольких процессов, и разрешить ее бывает не так просто, особенно если учесть, что условие для наступления блокировки наступает случайно и может выполняться достаточно редко. Это приводит к трудности в обнаружении причины остановки программы - все выглядит так, как будто программа работает, а потом, изредка, без всяких видимых причин зависает.

Задание

Выполняется при самостоятельной подготовке:

Изучить теоретическую часть описания ЛР и материал соответствующих лекций.

Выполняется в лаборатории:

1. Включить компьютер. Если на компьютере установлена операционная система Windows 95/98 - запустить ее в режиме эмуляции DOS. Запустить Турбо Паскаль 7.0.

2. Загрузить исходный текст примера TASKSWAP.PAS, изучить логику использования предлагаемой программной модели планировщика.

3. Загрузить исходный текст модуля планировщика H314TASK.PAS, изучить логику реализации планировщика.

4. Смоделировать ситуацию взаимной блокировки, написав для этого программу, использующую модуль H314TASK. Отладить и запустить программу, продемонстрировать результаты работы преподавателю.

5. Написать и отладить программу по индивидуальному заданию (см. ниже). Продемонстрировать результаты работы преподавателю.

6. Завершить работу в Турбо Паскале. Выключить компьютер.

Варианты индивидуальных заданий

Написать программу, выполняющую в режиме разделения времени указанные в таблице три процесса. Расшифровка потоков приводится ниже:

Вариант	Многопоточность	Поток 1	Поток 2	Поток 3
1	вытесняющая	A	C	E

2	вытесняющая	B	C	F
3	вытесняющая	A	D	F
4	вытесняющая	B	D	E
5	вытесняющая	A	B	D
6	невытесняющая	B	C	E
7	невытесняющая	A	C	F
8	невытесняющая	B	D	F
9	невытесняющая	A	D	E
10	невытесняющая	B	A	C

A. Поток ожидает нажатия клавиши Esc (#27), при ее нажатии должны завершаться все потоки. См. функции KeyPressed и ReadKey.

B. Поток завершается спустя количество секунд, равное номеру варианта, умноженному на 5. При его завершении должны завершаться все потоки.

Примечание: Использовать функцию Delay Паскаля для организации задержки нельзя, так как она блокирует прерывания.

C. Поток рисует на экране линии со случайными координатами обоих концов и цветом.

D. Поток рисует на экране линии от конца предыдущей нарисованной линии до точки со случайными координатами и цветом.

E. Поток рисует на экране 1000 точек со случайными координатами одного, общего для этой итерации, случайного цвета, потом рисует на их месте черные точки (стирает их), потом цикл повторяется.

F. Поток рисует на экране эллипсы случайного цвета, размера и положения.

Лабораторная работа №3

Изучение базового шаблона программы для Windows. Принцип отображения информации в окнах.

Работа выполняется в системе программирования Borland Delphi 2.0 в среде Windows 95. Использование средств визуального программирования и классов VCL не допускается, то есть Delphi используется просто как 32-разрядный компилятор с языка Паскаль для операционной системы Windows.

Теоретические положения

Любая программа Windows при своем запуске получает от операционной системы ряд параметров. Эти параметры могут быть проанализированы внутри программы. Для доступа к ним в Паскаль-программе следует использовать следующие идентификаторы:

hInstance: Longint — Хэндл экземпляра приложения. Это число идентифицирует процесс, который представляет собой запущенная программа, в операционной системе. Хэндл экземпляра требуется некоторым функциям API.

CmdLine: PChar — Командная строка, с помощью которой запущено приложение. Содержит полное имя исполняемого модуля, а также параметры командной строки.

CmdShow: integer — Числовая константа, определяющее, каким должно быть показано главное окно программы (см. описание функции ShowWindow). Этот параметр может игнорироваться приложением.

Как правило, программы, написанные для Windows, используют окна для организации пользовательского интерфейса. Из этого правила могут быть исключения, но в подавляющем большинстве случаев программисту приходится работать с окнами.

Основой оконного графического пользовательского интерфейса Windows является механизм сообщений. Когда с окном должно быть произведено некоторое действие, операционная система посылает окну сообщение соответствующего содержания, которое это окно должно обработать. Например, в качестве реакции на сообщение WM_HIDE окно должно исчезнуть с экрана (сделаться невидимым). На самом деле посылка сообщения приводит к вызову некоторой процедуры ("оконной процедуры"), в качестве параметра которой передается небольшой блок данных, который и называется сообщением.

Сообщения могут передаваться оконной процедуре немедленно, а могут помещаться в очередь сообщений, откуда программа должна выбирать их по мере обработки предыдущих сообщений. Хотя сообщения адресуются ОКНАМ, Windows создает очередь сообщений для каждого ПРОЦЕССА (точнее — для каждого потока выполнения, их в рамках одной задачи, т.е. процесса, может быть несколько). Когда в каком-либо потоке программы порождается окно, Windows фиксирует его принадлежность к этому потоку и помещает сообщения для этого окна в очередь сообщений породившего его потока. В простейшем случае, когда в программе не организуются несколько потоков выполнения, очередь сообщений — одна на всю программу, и все сообщения, адресуемые окнам программы, помещаются в эту очередь, откуда программа должна уметь их забирать.

Из очереди сообщения извлекаются в виде блоков данных, структура которых такова:

```
TMsg = packed record
    hwnd: HWND; // хэндл окна-адресата
```

```
    message: UINT; // код (тип) сообщения
    wParam: WPARAM; // параметр сообщения
    lParam: LPARAM; // параметр сообщения
    time: DWORD; // момент посылки сообщения
    pt: TPoint; // положение курсора мыши в момент посылки сообщения
  end;
```

Сообщения содержит в себе информацию о том, какому окну оно адресовано (хэндл окна в поле hwnd), идентификатор сообщения, определяющий тип требуемого действия (поле message), и два параметра, назначение которых специфично для каждого типа сообщений. В ранних версиях Windows поле wParam было 16-разрядным, поле lParam - 32-разрядным, однако в Windows 95 (Win32) оба эти параметра 32-разрядные.

Тип сообщения задается целым числом в поле Message. В Windows определены более 1000 стандартных сообщений; кроме того, программист может вводить дополнительные сообщения. Все сообщения Windows перечислены в файле MESSAGES.PAS в качестве числовых констант. В любом трансляторе для Windows используется устоявшаяся система идентификаторов для обозначения типов сообщений, например: WM_CLOSE, WM_PAINT и т.д. При этом нет необходимости помнить, что WM_CLOSE=10h, WM_PAINT=0Fh и т.д.

В тело сообщения включается также дополнительная информация: о времени посылки сообщения и положении курсора мыши в этот момент. Нетрудно видеть, что размер сообщения фиксирован и вся "полезная информация" должна укладываться в 8 байт (поля wParam и lParam). Если сообщение подразумевает необходимость передачи большего объема информации, то передается указатель на блок памяти, содержащий эту информацию, и откуда процедура, обрабатывающая сообщение, должна эту информацию прочитать.

Практически в любой программе, работающей с окнами Windows, можно логически выделить следующие разделы:

1. Порождение окон.
2. Цикл выборки сообщений из очереди.
3. Оконные процедуры, реализующие логику обработки сообщений в окнах программы.

Вся смысловая нагрузка программы, все ее прикладное значение обычно сосредоточено в третьем из перечисленных разделов, точнее — в процедурах, которые вызываются внутри оконной процедуры. Почти все, что делает программа в Windows, она делает в ответ на сообщение, свидетельствующее о каком-либо внешнем событии. Пользователь щелкнул мышью по кнопке — окно получило сообщение WM_COMMAND — начали выполняться какие-то действия. Пользователь закрыл окно — все окна, которые были под только что закрытым окном на экране получили сообщение WM_PAINT и должны перерисоваться.

Общая структура шаблона программы

В данной работе предлагается составить программу, соответствующую следующему шаблону:

```
uses Messages, Windows;

procedure WndProc(...); stdcall; //Оконная процедура
begin
  case
    //..................... обработка WM_PAINT
```

```
      else
        result:=DefWindowProc(...);
    end; //case
  end;

  begin
    {регистрация оконного класса}
    {порождение окна}
    {вывод окна на экран}
    {цикл обработки сообщений}
  end.
```

Оконные процедуры

Оконная процедура в Delphi описывается как функция, возвращающая 32-битовое целое значение, с использованием директивы **STDCALL.** STDCALL указывает компилятору, что вызов такой функции происходит по правилам вызова процедур системой Windows. Набор параметров оконной процедуры следующий:

```
function WndProc1(hWnd: THandle; Msg: integer;
                  wParam: longint; lParam: longint
): longint; stdcall;
```

Оконная процедура реализует логику обработки сообщений окном и вызывается операционной системой всякий раз, когда окну передается сообщение для обработки. Нетрудно видеть, что параметры оконной процедуры соответствуют полям структуры сообщения, определяющим суть сообщения. Это хэндл окна-получателя, типа сообщения и параметры wParam и lParam. Результат, возвращаемый оконной процедурой, зависит от типа обрабатываемого сообщения. Для некоторых сообщений необходимо вернуть определенный результат, например, чтобы сообщить отправившему сообщение процессу, что оно успешно обработано. Для других сообщений результат не важен, в этом случае рекомендуется возвращать 0.

Должна ли оконная процедура уметь обрабатывать все 1000 с лишним возможных сообщений? С одной стороны — да, ведь все сообщения, адресованные окну, проходят через его оконную процедуру. Но с другой стороны — в Windows определена реакция по умолчанию на все возможные сообщения, поэтому подавляющее большинство сообщений оконная процедура, которую пишет программист, может поручить обработать Windows, вызвав внутри себя

```
DefWindowProc(hwnd,msg,wparam,lparam);
```

Поэтому тело оконной процедуры обычно представляет собой один большой оператор **case**, завершаемый командой "а иначе выполнить обработку по умолчанию". Например так:

```
case Message of
  WM_PAINT: doPaint;     // вызов процедуры реакции на сообщение
  WM_DESTROY: PostQuitMessage(0); // приведет к завершению программы
  else
    result:=DefWindowProc(hwnd,msg,wparam,lparam); // обработка по умолчанию
end; // case
```

Регистрация оконных классов

Для порождения окна прежде всего должен быть описан класс окна. Класс окна — это некоторый блок памяти, существующий в недрах Windows, в котором содержится описание различных параметров, определяющих внешний вид и поведение окон (экземпляров данного

класса), и в частности — указатель на оконную процедуру, обрабатывающую сообщения любого окна данного класса. Как уже понятно из вышесказанного, одновременно может существовать множество окон - экземпляров одного и того же класса, со сходными свойствами и поведением, и окно в Windows всегда порождается как экземпляр некоторого зарегистрированного в системе класса. В Windows имеется несколько предописанных оконных классов, однако обычно программисту приходится описывать и регистрировать свои собственные классы окон.

Для регистрации оконного класса программа должна вызвать функцию

```
function RegisterClassEx(const WndClass: TWndClassEx): ATOM;
```

В случае успешной регистрации класса функция возвращает ненулевое целое значение, которое можно использовать для идентификации оконного класса в других процедурах API; нулевой результат свидетельствует о невозможности создания класса. Параметр WndClass описывает необходимые для создания класса параметры и имеет следующую структуру:

```
TWndClassEx = packed record
    cbSize: UINT; // размер структуры TWndClassEx
    style: UINT;  // битовые флаги, определяющие стиль окна
    lpfnWndProc: TFNWndProc; // адрес оконной процедуры
    cbClsExtra: Integer; // Размер дополнительной памяти класса
    cbWndExtra: Integer; // Размер дополнительной памяти окна
    hInstance: HINST;  // Хэндл экземпляра приложения, которому принадлежит
                       // оконная процедура
    hIcon: HICON;  // Хэндл значка окна (32x32 пиксела)
    hCursor: HCURSOR; //  Хэндл курсора окна по умолчанию
    hbrBackground: HBRUSH; // Хэндл кисти окна по умолчанию или цвет фона
    lpszMenuName: PAnsiChar; //  Оконное меню (имя ресурса)
    lpszClassName: PAnsiChar; // Строка - имя класса, заканчивается #0
    hIconSm: HICON; // Хэндл маленького значка окна (16x16 пикселов)
  end;
```

cbSize должно содержать размер передаваемого блока данных, т.е. SizeOf(TWndClassEx).

style является комбинацией битовых флагов, получаемых при логическом сложении (операция OR) следующих констант (приведены не все возможные стили):

CS_CLASSDC	Один контекст устройства выделяется для всех окон класса и используется при операциях рисования в них. При этом программист должен заботиться о том, чтобы операции рисования в окнах этого класса не происходили в разных потоках одновременно.
CS_DBLCLKS	Окна при двойном щелчке мышью внутри окна получают соответствующие сообщения.
CS_GLOBALCLASS	Позволяет объявить глобальный класс окна (актуально для DLL).
CS_HREDRAW	Перерисовывает все окно, если изменяется горизонтальный размер окна.
CS_NOCLOSE	Убирает команду "Закрыть" из системного меню окна и запрещает кнопку закрытия окна в его правом верхнем углу.
CS_OWNDC	Заранее выделяет уникальный контекст устройства для каждого окна класса..
CS_PARENTDC	Дочернее окно использует контекст устройства окна-родителя. В ряде случаев позволяет ускорить отрисовку дочерних окон.
CS_SAVEBITS	Окно при своем появлении сохраняет образ закрываемой им части экрана в буфере, а при скрытии восстанавливает этот образ из буфера; при этом тем окнам, что были под ним, не посылается сообщение WM_PAINT и они не перерисовываются. Этот стиль обычно используется для небольших по размеру окон, которые появляются на экране на короткое время (диалоги, меню).
CS_VREDRAW	Перерисовывает все окно, если изменяется вертикальный размер окна.

cbClsExtra и cbWndExtra указывают размер дополнительной памяти, резервируемой Windows для всего класса и каждого окна соответственно. Программа может обращаться к этой памяти и использовать ее для своих целей с помощью функций GetClassLong, GetClassWord, GetWindowLong, GetWindowWord, SetClassLong, SetClassWord, SetWindowLong, SetWindowWord (см. Win32 API Help).

hInstance должно содержать хэндл экземпляра приложения, в котором расположена оконная процедура. Инициализируется как

```
wndClass.hInstance:=hInstance;
```

В поле lpszClassname должен быть указатель на Z-строку (оканчивающуюся нулем), в которой содержится имя класса. Помните, что в Delphi тип String может означать как длинные строки, которые можно передавать в качестве Z-строк (ANSIString), так и короткие строки в стиле языка Паскаль предыдущих версий (ShortString), когда первый байт строки содержит ее длину, а нулевого символа в конце нет. Трактовка типа String определяется опцией компилятора {$H+/-}.

В оставшихся нерассмотренными полях указываются хэндлы ресурсов (значки, курсор), которые используются окнами класса по умолчанию. Если эти параметры не указывать (сделать нулевыми), то программа должна сама заботиться об изменении вида курсора при вводе указателя мыши в область окна, о рисовании значка при минимизации окна и т.д.

В системе Windows имеется небольшой набор предопределенных ресурсов и объектов GDI, которые можно использовать при описании классов окон. Их хэндлы можно получить с помощью функций

```
function LoadIcon(hInstance: HINST; lpIconName: PChar): HICON;
function LoadCursor(hInstance: HINST; lpCursorName: PChar): HCURSOR;
function GetStockObject(Index: Integer): HGDIOBJ;
```

При этом при загрузке предопределенного ресурса в функции LoadIcon и LoadCursor передается нулевое значение hInstance, а вместо имени ресурса передается индекс из числа предопределенных констант, например

```
wndClass.hCursor:=loadCursor(0, idc_Arrow);
```

(использовать в качестве курсора стандартный системный курсор в виде стрелки).

Классы окон, созданные приложением, уничтожаются после его завершения автоматически.

Создание окон

Зарегистрировав таким образом класс окна, программа может приступить к созданию самих окон. Создание окна производится функцией

```
function CreateWindow(
  lpClassName: PChar; // Имя зарегистрированного класса окна
  lpWindowName: PChar; // Заголовок окна
  dwStyle: DWORD;  // Стиль окна (комбинация флагов)
  X, Y, nWidth, nHeight: Integer;  // Начальное положение и размеры окна
  hWndParent: HWND; // Хэндл окна-родителя
  hMenu: HMENU;  // Хэндл меню окна
  hInstance: HINST;  // Экземпляр приложения, создающий окно
  lpParam: Pointer // Указатель на параметры для WM_CREATE
): HWND;
```

Существует также функция CreateWindowEx, среди параметров которой присутствует дополнительное поле флагов dwExStyle. остальные параметры и действия функции идентичны рассматриваемой.

Функция CreateWindow или CreateWindowEx создает окно и возвращает в случае успеха операции ненулевой хэндл созданного окна. В противном случае возвращается 0. Хэндл окна используется для идентификации окна при всех последующих операциях с ним. Рассмотрим подробнее некоторые ее параметры.

В качестве X, Y, nWidth и nHeight может быть передано значение CW_USEDEFAULT, при этом начальное положение окон и размеры будут определяться Windows.

Указание ненулевого хэндла родительского окна hWndParent заставляет порождаемое окно автоматически выполнять некоторые дополнительные действия, например: когда родительское окно минимизируется, подчиненные окна скрываются, подчиненные окна всегда располагаются на экране поверх родительского и т.д.

В поле dwStyle может указываться логическая сумма (OR) следующих флагов (указаны не все флаги):

WS_BORDER	Окно имеет границу в виде тонкой линии.
WS_CAPTION	Окно имеет границу и заголовок.
WS_CHILD	Окно является дочерним, т.е. отображается в клиентской области родительского окна.
WS_CLIPCHILDREN	Области, занятые дочерними окнами, не перерисовываются в родительском окне. Используется при создании родительских окон.
WS_CLIPSIBLINGS	Области, занятые перекрывающими данное дочернее дочерними окнами не перерисовываются. Используется при создании дочерних окон.
WS_DISABLED	Создает окно, в запрещенном состоянии, т.е. не воспринимающим событий от мыши и клавиатуры.
WS_DLGFRAME	Обрамление окна в стиле диалога. Окно не может иметь флага заголовка.
WS_HSCROLL	Окно имеет горизонтальную полосу прокрутки.
WS_ICONIC, WS_MINIMIZE	Окно создается минимизированным.
WS_MAXIMIZE	Окно создается максимизированным.
WS_MAXIMIZEBOX	Окно имеет кнопку максимизации.
WS_MINIMIZEBOX	Окно имеет кнопку минимизации.
WS_OVERLAPPED, WS_TILED	Создает экранное окно, имеющее заголовок и рамку.
WS_OVERLAPPEDWINDOW, WS_TILEDWINDOW	Комбинация WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBOX, WS_MAXIMIZEBOX.
WS_POPUP	Создает всплывающее окно. Несовместимо с WS_CHILD.
WS_POPUPWINDOW	Комбинация WS_BORDER, WS_POPUP, WS_SYSMENU. Чтобы системное меню было видимым, должны быть указаны стили WS_CAPTION и WS_POPUPWINDOW.
WS_SIZEBOX, WS_THICKFRAME	Окно с рамкой для изменения размеров.
WS_SYSMENU	Окно имеет системное меню в левом верхнем углу. Должен присутствовать флаг WS_CAPTION (заголовок).
WS_VISIBLE	Окно изначально видимо на экране.
WS_VSCROLL	Окно имеет вертикальную полосу прокрутки.

Замечание. Стиль WS_OVERLAPPED принимается по умолчанию (значение этой константы равно нулю). При этом управление видом рамки и наличием заголовка не работает (они есть всегда, и рамка всегда "толстая"). Для создания независимых (не-дочерних) окон с полностью управляемым внешним видом используйте WS_POPUP.

В параметре dwExStyle функции CreateWindowEx может указываться логическая сумма (OR) следующих флагов (указаны не все флаги), определяющих дополнительный стиль окна:

WS_EX_ACCEPTFILES Окно поддерживает перенос файлов drag-and-drop.

WS_EX_APPWINDOW Windows 95: Значок окна отображается в панели задач.

WS_EX_CLIENTEDGE Рамка такова, что окно выглядит "утопленным".

WS_EX_CONTROLPARENT Windows 95: Поддержка автоматического переключения между дочерними окнами управления по клавише TAB.

WS_EX_DLGMODALFRAME Окно с широкой рамкой диалога. dwStyle может содержать WS_CAPTION.

WS_EX_MDICHILD Дочернее окно многооконного интерфейса MDI.

WS_EX_OVERLAPPEDWINDOW Windows 95: Комбинация WS_EX_CLIENTEDGE и WS_EX_WINDOWEDGE.

WS_EX_PALETTEWINDOW Windows 95: Окно палитры. Комбинация WS_EX_WINDOWEDGE, WS_EX_TOOLWINDOW и WS_EX_TOPMOST.

WS_EX_STATICEDGE Windows 95: Трехмерная рамка для окон, не предназначенных для ввода информации пользователем.

WS_EX_TOOLWINDOW Windows 95: Инструментальное окно с уменьшенным заголовком, не появляется в панели задач или в списке, появляющемся по ALT+TAB.

WS_EX_TOPMOST Окно переднего плана. Изображается поверх всех окон, не имеющих этого атрибута.

WS_EX_TRANSPARENT Прозрачное окно: все окна находящиеся под ним, получают сообщение WM_PAINT и прорисовываются, после чего сообщение WM_PAINT получает данное окно.

WS_EX_WINDOWEDGE Windows 95: Рамка такова, что окно выглядит выпуклым.

Окно показывается или убирается с экрана при помощи функции

```
function ShowWindow(hWnd: HWND; nCmdShow: Integer): BOOL;
```

В параметре nCmdShow указывается действие, которое необходимо произвести с окном, хэндл которого hWnd. В частности, SW_SHOW означает "показать окно".

Организация обработки очереди сообщений

Сообщения, которые посылаются окнам с буферизацией (post), в отличие от сообщений, посылаемых без буферизации (send), передаются операционной системой не напрямую в оконную процедуру окна-получателя, а помещаются в очередь сообщений. Эта очередь организуется для каждого потока, в котором порождались окна. Если в программе не организовано множество потоков, то можно говорить о единой очереди сообщений программы. Выбор сообщений из очереди и передача их на обработку в соответствующие оконные процедуры возложены на само приложение.

В простейшем случае выбор сообщений организуется в виде следующего цикла:

```
while GetMessage(msg,0,0,0) do begin {получить очередное сообщение}
  TranslateMessage(msg);   {Windows транслирует сообщения от клавиатуры}
  DispatchMessage(msg);    {Windows вызовет оконную процедуру}
end; {выход по wm_quit, на которое GetMessage вернет FALSE}
```

Функция GetMessage с указанными параметрами извлекает очередное сообщение из очереди и помещает его в структуру msg типа TMsg. Если сообщений в очереди нет, то

текущий поток переводится в состояние ожидания и досрочно отдает управление другим потокам. Функция GetMessage возвращает истину при получении любого сообщения, кроме WM_QUIT.

Сообщение WM_QUIT свидетельствует о том, что либо пользователь, либо операционная система подали приложению команду завершиться. При получении сообщения WM_QUIT цикл выборки сообщений завершается и программа, возможно, выполнив какие-то завершающие действия, также завершается.

Любое другое сообщение, отличное от WM_QUIT, приводит к результату TRUE в функции GetMessage и должно быть передано в оконную процедуру. Это выполняется функцией DispatchMessage, при выполнении которой Windows анализирует данные сообщения и вызывает нужную оконную процедуру.

Обычно перед передачей сообщения оконной процедуре выполняются дополнительные действия, самое распространенное из которых — обработка сообщений от клавиатуры функцией TranslateMessage. Эта функция при получении низкоуровневого сообщения о нажатии клавиши WM_KEYDOWN в случае нажатия алфавитно-цифровой клавиши помещает в очередь сообщений сообщение WM_CHAR, содержащее вместо кода клавиши код соответствующего символа в зависимости от установленного в системе языкового драйвера. В дальнейшем это сообщение выбирается из очереди и обрабатывается на общих основаниях.

Посылка сообщений внутри программы

Посылка сообщений окнам не есть прерогатива операционной системы. Любой пользовательский процесс также может посылать сообщения. Для этого есть две основные функции Windows API, реализующие буферизованную и небуферизованную посылку сообщений.

Под буферизованной посылкой сообщений понимается помещение их в очередь того потока, в котором создавалось окно-получатель. Эта операция реализуется функцией

```
function PostMessage(hWnd: HWND; Msg: UINT;
                     wParam: WPARAM; lParam: LPARAM): BOOL;
```

Сообщение типа Msg с параметрами wParam и lParam отправляется окну с хэндлом hWnd. Функция возвращает TRUE, если сообщение отправлено успешно, и FALSE в противном случае. Важно, что после выполнения функции PostMessage, сообщение помещено в очередь, но еще не обработано оконной процедурой окна-получателя.

Посылка сообщения без буферизации выполняется функцией

```
function SendMessage(hWnd: HWND; Msg: UINT;
                     wParam: WPARAM; lParam: LPARAM): LRESULT;
```

При этом сообщение не помещается в очередь, а вместо этого непосредственно вызывается оконная процедура окна-приемника, которая обрабатывает сообщение. Значение, возвращаемое оконной функцией, возвращается в качестве результата функции SendMessage. Таким образом, после возврата из функции SendMessage посланное сообщение уже обработано и известен результат его обработки.

Организация отображения информации в окнах

Для операционной системы DOS были разработаны приложения и оконные библиотеки для текстового режима, в которых при открытии окна закрываемая им область

экрана сохранялась в особом буфере в памяти, а после закрытия окна восстанавливалась из этого буфера. Таким образом, окно обязано было заботиться о восстановлении экрана после своего исчезновения.

Метод сохранения и восстановления части экрана, закрываемой окном, в буфере неудобен для построения графического интерфейса, так как слишком велик объем информации, который требовалось бы сохранять: в полноцветном режиме, когда в видеопамяти отводится 24 или 32 бита на каждый пиксел, буфер для окна, занимающего весь экран, требовал бы более мегабайта оперативной памяти, что нереально. Поэтому в основе графического интерфейса Windows лежит другой принцип: каждое окно в любой момент времени должно уметь отображать себя само. Операционная система при этом отслеживает, какие участки каких окон требуют перерисовки и посылает этим окнам сообщения WM_PAINT, обрабатывая которые, оконные процедуры этих окон должны обеспечить отображение информации в окне.

Это накладывает на программиста некоторые дополнительные ограничения. Так, в программах для DOS программист мог один раз вывести на экран изображение и быть уверенным, что оно останется в целости и сохранности, пока сама программа его не сотрет. В то время как в Windows изображение в окне может быть стерто в силу внешних причин, например, когда другое окно закрыло окно программы, а затем было убрано с экрана. Поэтому данные, необходимые для прорисовки изображения в окне (редактируемый текст или массив статистических данных для диаграммы) должны быть наготове всегда, пока окно видимо на экране.

Обработка сообщения WM_PAINT

Когда операционная система считает, что окно или его часть должны быть перерисованы, оно посылает ему сообщение WM_PAINT, не имеющее параметров. Обработка сообщения WM_PAINT собственно и должна состоять в перерисовке изображения в окне. Оконная процедура при обработке WM_PAINT должна возвращать 0.

Для рисования в программе для Windows используются функции GDI (графического интерфейса устройства), являющиеся унифицированным интерфейсом драйверов дисплея, а также принтеров, графопостроителей и прочих устройств вывода графической информации. Чтобы воспользоваться большинством из этих функций, необходимо получить так называемый "контекст устройства". Фактически, это внутренняя структура данных GDI, содержащая такие сведения, как возможная глубина цвета, разрешающая способность устройства, способ пересчета логических координат в физические, а также об активных инструментах рисования, таких как перо, кисть и шрифт.

Часто требуется перерисовать не все окно целиком, а некоторую его часть, допустим, когда два окна частично перекрывались, и затем одно из них было закрыто. Поэтому при получении сообщения WM_PAINT определен так называемый недействительный регион, в пределах которого требуется производить рисование. Чаще всего недействительный регион является совокупностью прямоугольных областей. Также существует понятие региона отсечения, за пределами которого рисование не производится. Чаще всего регион отсечения по координатам совпадает с недействительным регионом, но это разные понятия.

Обработка сообщения WM_PAINT в оконной процедуре должна строиться примерно следующим образом:

```
case message of
  .......
  WM_PAINT: begin
      hdc:=BeginPaint(hwnd, paintStruct);
```

```
        {вызовы функций GDI}
        EndPaint(hwnd, paintStruct);
      end;
    ..............
end; //case
```

Функция BeginPaint возвращает хэндл контекста устройства для окна с хэндлом hWnd, кроме того, она заполняет структуру paintStruct некоторой полезной информацией. Помимо этих действий, функция BeginPaint устанавливает регион отсечения в соответствии с недействительным регионом, а сам недействительный регион делает действительным. Вызов BeginPaint должен производиться только при обработке сообщения WM_PAINT и обязательно должен сопровождаться вызовом функции EndPaint. Параметр paintStruct должен быть записью типа

```
TPaintStruct = packed record
    hdc: HDC;
    fErase: BOOL;
    rcPaint: TRect;
    fRestore: BOOL;
    fIncUpdate: BOOL;
    rgbReserved: array[0..31] of Byte;
  end;
```

Важными для программиста являются первые три поля этой структуры. В поле hdc дублируется хэндл полученного контекста устройства. В поле fErase содержится признак того, была ли произведена операция стирания фона окна в пределах региона отсечения, это стирание производится с использованием кисти, указанной при создании класса окна, и выполняется Windows автоматически при вызове BeginPaint. Наконец, rcPaint содержит координаты минимальной прямоугольной области, описанной вокруг региона отсечения.

Смысл параметра rcPaint в том, что за пределами этого прямоугольника рисование гарантированно не производится. В то же время, так как регион отсечения может иметь сложную форму, операции рисования могут физически не производиться и в некоторых областях внутри этого прямоугольника. Так как выполняется отсечение, обработчик WM_PAINT может просто перерисовывать окно, ни о чем не заботясь, так как все равно будет прорисована только часть изображения внутри региона отсечения. Тем не менее, использование знаний о размерах региона отсечения позволяет оптимизировать процесс рисования за счет того, что части изображения, лежащие заведомо вне rcPaint, можно просто не рисовать.

Дисциплина ожидания в очереди для сообщения WM_PAINT является особой: это сообщение имеет низший приоритет и может быть получено только в том случае, когда в очереди нет других сообщений. Кроме того, если в очереди уже присутствует сообщение WM_PAINT и приходит новое сообщение WM_PAINT для того же окна, то это приводит просто к добавлению нового региона обновления к региону обновления, соответствующему старому , еще не обработанному, сообщению WM_PAINT. При этом в очереди по-прежнему останется одно сообщение WM_PAINT. Наконец, если в очереди есть сообщение WM_PAINT, но соответствующий регион обновления каким-то образом стал корректным (например, приложение вызвало функцию ValidateRect), то сообщение WM_PAINT будет удалено из очереди, так как рисование больше не требуется. Эту логику прохождения сообщения WM_PAINT обеспечивает операционная система.

Функции рисования GDI

(Конкретные особенности использования функций см. в Help.)

Функция	**Примечание**

GetClientRect	Позволяет получить размеры клиентской области окна
MoveToEx	Перемещение текущей позиции пера
LineTo	Рисование отрезка из текущей позиции и перемещение пера
Ellipse	Рисование закрашенного эллипса, вписанного в указанный прямоугольник
Arc	Рисование дуги эллипса, вписанного в указанный прямоугольник
Rectangle	Рисование закрашенного прямоугольника
RoundRect	Рисование закрашенного прямоугольника со скругленными краями
PolyLine	Рисование ломаной, соединяющей указанные точки
Polygon	Рисование заполненного замкнутого многоугольника по заданным вершинам
TextOut	Выводит строку символов
DrawText	Выводит строку символов с расширенными возможностями форматирования
SetBkMode	Управляет "прозрачностью" текста
SetTextAlign	Устанавливает режим вывода текста
GetPixel	Прочитать цвет пиксела
SetPixel	Нарисовать пиксел

Задание

Выполняется при самостоятельной подготовке:

Изучить теоретическую часть описания ЛР и материал соответствующих лекций.

Выполняется в лаборатории:

1. Включить компьютер. Запустить систему Delphi 2.0.

2. Загрузить исходный текст примера LAB3.PAS, а также модули WINDOWS.PAS и MESSAGES.PAS, изучить логику построения программы для Windows.

3. Откомпилировать и запустить пример. Изучить поведение созданного окна.

4. Написать и отладить программу по индивидуальному заданию (см. ниже). Продемонстрировать результаты работы преподавателю.

5. Завершить работу в Delphi. Оставить компьютер включенным.

Варианты индивидуальных заданий

№	Задание
1.	Стандартное масштабируемое окно, в котором по центру нарисованы Декартовы оси координат с метками 0 и названием осей X и Y, размеры осей должны изменяться при изменении размеров окна.
2.	Масштабируемое окно без заголовка, в котором по центру нарисованы Декартовы оси координат с метками 0 и названием осей X и Y, размеры осей должны изменяться при изменении размеров окна.
3.	Два окна без заголовков. Программа должна завершаться после закрытия всех окон.
4.	"Главное окно" с заголовком и масштабируемое, а также два "вспомогательных" окна стиля Toolbar

(плавающая панель инструментов). Программа должна завершаться после закрытия главного окна.

5. Стандартное масштабируемое окно, в левом верхнем углу которого нарисованы Декартовы оси координат с метками 0 и названием осей X и Y, размеры осей должны составлять 200 пикселов по горизонтали и вертикали.
6. Масштабируемое окно без заголовка, в правом нижнем углу которого нарисованы Декартовы оси координат с метками 0 и названием осей X и Y, размеры осей должны составлять 200 пикселов по горизонтали и вертикали.
7. Два стандартных окна. Программа завершается при закрытии любого окна. При минимизации одного окна другое должно максимизироваться.
8. Два стандартных окна. Программа завершается при закрытии любого окна. При минимизации одного окна другое должно изменять цвет клиентской области на инверсный (с белого на черный и наоборот).
9. Родительское окно и два дочерних, окна стандартного вида, масштабируемые. В дочерних окнах по центру отображаются Декартовы оси координат с метками осей протяженностью 200 пикселов. Программа завершается при закрытии родительского окна.
10. Родительское окно и два дочерних, окна стандартного вида, масштабируемые. В дочерних окнах по центру отображаются Декартовы оси координат с метками осей, протяженность осей изменяется при изменении размеров окна. Программа завершается при закрытии родительского окна.

Лабораторная работа № 4

Изучение возможностей GDI для отображения информации в окнах Windows

Работа выполняется в системе программирования Borland Delphi 2.0 в среде Windows 95. Использование средств визуального программирования и классов VCL не допускается, то есть Delphi используется просто как 32-разрядный компилятор с языка Паскаль для операционной системы Windows.

Теоретические положения

Графический интерфейс устройства (GDI) — это подсистема Windows, отвечающая за отображение графической информации на видеодисплеях и принтерах. Наряду с двумя другими основными подсистемами (KERNEL — ядро, управление памятью, обеспечение многозадачности, файловый ввод/вывод, и USER — организация оконного пользовательского интерфейса) GDI является важнейшей составной частью Windows.

GDI поддерживает аппаратно-независимую графику. Можно рассматривать GDI как высокоуровневый интерфейс для аппаратных средств графики, как своего рода графический язык программирования. Схема прохождения команд управления между программой и аппаратурой отображения в Windows представлена на рис.1. При отображении информации программа обязательно должна пользоваться средствами GDI, попытки осуществить вывод напрямую в видеопамять, минуя GDI, будут пресекаться операционной системой, и даже если ее удастся "обмануть", никто не гарантирует, что такая программа будет работоспособной в будущих версиях Windows.

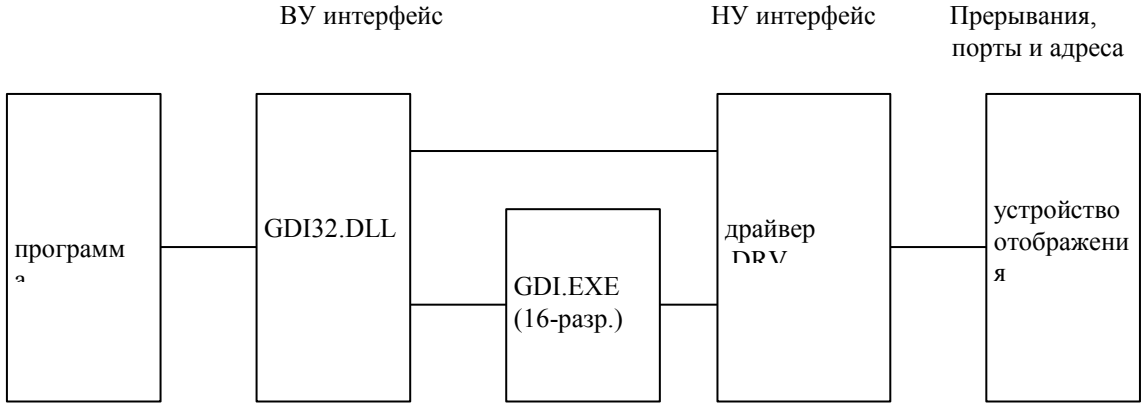


Рис.1 Прохождение команд между программой и устройством отображения

GDI состоит из нескольких сотен функций, описание которых можно получить, воспользовавшись справочной системой по Win32 API. Далее будут освещены лишь ключевые моменты, касающиеся основных вопросов применения функций GDI. Содержащиеся далее утверждения верны "в большинстве случаев", т.е. в некоторых ситуациях они могут не соответствовать действительности. Однако описание таких ситуаций потребовало бы во много раз большего места, чем приводимое описание, поэтому опускается.

Группы функций

Функции GDI можно разделить на следующие крупные группы:

- Функции получения и освобождения контекста устройства, такие как BeginPaint, EndPaint, GetDC, ReleaseDC и др. Понятие контекста устройства рассмотрено в теоретическом материале к ЛР2.

- Функции получения информации о контексте устройства, например GetTextMetrics.
- Функции рисования линий, эллипсов, закрашенных областей и текста.
- Функции управления атрибутами контекста устройства, такие как SetBkMode, SettextAlign, и соответствующие им функции "Get".
- Функции для работы с объектами GDI, такими как перо, кисть, шрифт.

Примитивы GDI

Типы графических объектов, выводимых на устройство вывода, могут быть разделены на несколько категорий, называемых "примитивами". К ним относятся:

- Отрезки прямых и кривые. GDI поддерживает рисование прямых (отрезков), прямоугольников, эллипсов и окружностей, дуг эллипсов, сплайнов Безье, а также ломаных линий, состоящих из множества отрезков. Линии рисуются с использованием объекта GDI "перо" (Pen).
- Закрашенные области. Области закрашиваются с использованием объекта GDI "кисть" (Brush).
- Битовые образы. Битовый образ — это двумерный набор битов, соответствующих пикселам устройства отображения. Для работы с битовыми образами используются соответствующие объекты GDI (Bitmap).
- Текст. GDI поддерживает как растровые шрифты, так и шрифты TrueType, являющиеся векторными и представляющие собой информацию о контурах символов, который закрашивается определенным цветом. GDI поддерживает вывод текста не только по горизонтали, но под любым углом к горизонтали вдоль некоторой прямой. Вывод текста осуществляется с использованием объекта GDI "шрифт" (Font).

Концепция использования объектов GDI

Для управления выводом графических примитивов в контексте устройства должен быть выбран тот или иной объект GDI (шрифт, перо, кисть). Если в программе необходимо изменить, например, начертание шрифта или стиль заливки областей, необходимо в общем случае *создать* новый объект GDI (шрифт или кисть соответственно), *выбрать* его в текущем контексте устройства, произвести требуемые операции рисования, затем *освободить* объект из контекста устройства (при этом обычно снова выбирается объект, являвшийся выбранным до того) и *удалить* (уничтожить) объект.

Режимы масштабирования и преобразования

Вывод в функциях GDI производится в логических координатах. По умолчанию координаты вывода и размеры задаются в пикселах, однако существуют и другие возможности. Могут использоваться доли дюйма, доли миллиметра, типографские пункты, а также любые удобные для программиста единицы.

Другие важные понятия

Регион — это сложная область произвольной формы, возможно, не являющаяся непрерывной. Обычно регион задают как комбинацию простых фигур (например — прямоугольников). Следует понимать, что регион — это не то, что *отображается* на экране, это просто фигура в смысле координат. В то же время, регион можно визуализировать, обведя его контур или выполнив его заливку цветом. Внутреннее представление региона внутри GDI — это набор координат горизонтальных линий сканирования. Обычно регионы используются для задания области отсечения.

Путь (path) — это набор отрезков и кривых. Пути могут быть преобразованы в регионы, а также использованы для рисования, закрашивания и отсечения.

Отсечение (clipping). Рисование может быть ограничено в пределах некоторой области окна, задаваемой регионом или путем. Операции рисования приводят к изменению изображения только в пределах области отсечения; если, например, линия выходит за границы области отсечения, то она прорисовывается только внутри области вплоть до ее границы.

Контекст устройства

Работа с контекстом устройства при обработке сообщения WM_PAINT при помощи функций BeginPaint и EndPaint описана в теоретическом материале к ЛР2.

При необходимости выполнить рисование в других местах программы, т.е. в теле обработчика сообщения, отличного от WM_PAINT, следует пользоваться следующими командами:

```
hdc:=GetDC(hwnd);
{операции рисования}
ReleaseDC(hdc);
```

или

```
hdc:=GetWindowDC(hwnd);
{операции рисования}
ReleaseDC(hdc);
```

Функция GetDC возвращает хэндл контекста устройства для рабочей области окна, функция GetWindowDC — для всего окна (включая заголовок, рамку и полосы прокрутки). Функция GetWindowDC используется редко и полезна при нестандартной обработке сообщения WM_NCPAINT, которое оконная процедура получает при необходимости прорисовки заголовка и рамки, т.е. нерабочей области окна. После использования контекст устройства обязательно должен быть освобожден.

Функция GetDC(0) вернет хэндл контекста устройства для всего экрана.

Для получения контекста произвольного устройства системы используется функция CreateDC (см. Справку по Win32 API); в этом случае контекст устройства должен не освобождаться, а удаляться при помощи функции DeleteDC. Если контекст устройства нужен не для операций рисования, а лишь для получения инйормации о характеристиках устройства, следует использовать функцию CreateIC с параметрами, аналогичными CreateDC. Эта функция создает так называемый информационный контекст устройства, который также впоследствии должен быть удален при помощи вызова DeleteDC.

При работе с битовыми образами (или при необходимости проводить рисование в буфере памяти, что одно и то же) используется контекст памяти, *совместимый* с реальным контекстом устройства вывода. Для работы с ним используются следующие функции:

```
hdcMem:=CreateCompatibleDC(hdc);
{операции рисования в памяти}
DeleteDC(hdcMem);
```

Получение информации из контекста устройства может быть осуществлено различными функциями GDI, название которых как правило начинается с "Get". Самой общеупотребительной из них является

function GetDeviceCaps(hdc:THandle; iIndex:integer):integer;

возвращающая значение параметра, задаваемого константой в iIndex. Перечень возможных параметров можно получить, обратившись к Справке.

Цвет

Функции GDI используют 24-битовую модель цвета TrueColor. Так как трехбайтовых целых типов нет, то цвет задается четырехбайтовым целым числом (longint в 16-разрядных приложениях, а также integer в 32-разрядных) в следующем формате (речь идет о 16-ричных позициях):

00 BB GG RR

Младший байт задает интенсивность красной составляющей, первый байт — зеленой, второй — синей, старший байт нулевой. Для получения такого RGB-цвета из компонент в компиляторах определен макрос RGB(r,g,b) (в Delphi — функция RGB(r,g,b:integer):integer).

Если видеоадаптер работает в режиме с меньшим количеством цветов, то Windows при выводе на экран либо заменяет 24-битовый цвет на ближайший цвет, который может быть отображен видеоадаптером (при рисовании линий, при выводе фона текста), либо (при закрашивании областей в режимах с 16 и 256 цветами) производит растрирование (dithering), т.е. заполняет область не чистым цветом, а набором точек разных цветов, которые визуально воспринимаются как область, имеющая близкий к заданному цвет.

Можно определить ближайший к желаемому чистый цвет, отображаемый устройством, при помощи функции GetNearestColor, например

rgbPureColor:=GetNearestColor(hdc, rgbColor);

Рисование "до, не включая"

Список самых общеупотребительных функций рисования линий, областей и текста приведен в описании ЛР3. Здесь будут рассмотрены лишь принципы их применения.

Прежде всего, чтобы избегать ошибок типа "потерянная единица", надо помнить, что функции GDI рисуют по правилу "от и до, не включая конец". Это означает, например, что функция LineTo(x,y) проводит линию из текущей позиции пера в указанную точку (x,y), при этом текущая позиция пера и все точки линии закрашиваются, а точка (x,y) остается незакрашенной. Это удобно при рисовании ломанных линий в режиме с растровой операцией, отличной от прямого копирования, когда цвет пера объединяется с цветом фона по какому-либо закону — при этом не будет разрывов в точке сочленения отрезков.

Функции, использующие прямоугольные области, получают в качестве параметров координаты левого верхнего и правого нижнего углов прямоугольника. Другим проявлением рассматриваемого принципа является то, что при этом левый верхний угол нарисованного прямоугольника будет иметь указанные координаты, а правый нижний угол нарисованной фигуры всегда будет расположен на единицу *выше и левее* точки, указанной в качестве левого нижнего угла. (Здесь речь идет именно физически о правом верхнем и левом нижнем углах фигуры, а не о парах чисел, переданных в функцию рисования в качестве координат соответствующих углов.)

Наконец, координаты в Windows 95 API хотя и представляются 32-разрядными целыми числами, не могут выходить за границы 16-разрядного знакового диапазона, т.е. от -32768 до 32767. Windows NT такого ограничения не имеет.

Эллипсы и эллиптические дуги

Windows поддерживает только рисование эллипсов с горизонтально-вертикальным расположением осей.

Для рисования эллипсов функции GDI используют непривычный на первый взгляд прием: в качестве параметров функции передается не центр и размеры полуосей эллипса, а координаты левого верхнего и правого нижнего углов прямоугольника, *описанного* вокруг желаемого эллипса.

При рисовании дуг, а также секторов и сегментов (функции Arc, Chord и Pie), эллипс, которому принадлежит дуга, также задается описанным прямоугольником. Кроме того задаются секущие лучи, как бы проводимые из центра эллипса через две точки с указанными

координатами. Эти точки не обязательно должны лежать на линии эллипса. Дуга будет проведена от точки пересечения эллипса с первым лучом до точки пересечения эллипса со вторым лучом; в случае рисования сегмента или сектора будут проведены также хорда или два радиуса в полученные точки пересечения, а получившаяся фигура будет заполнена при помощи текущей кисти.

Этот способ задания эллиптических кривых крайне неудобен при необходимости работы с дугами, заданными в привычной полярной системе координат "угол-радиус".

Использование объектов GDI

Для операций рисования в текущем контексте устройства должны быть *выбраны* некоторые перо, кисть и шрифт. По умолчанию выбраны черное перо толщиной 1 пиксел, рисующее непрерывную линию, белая кисть и шрифт "System". При необходимости изменить эти параметры, необходимо в общем случае создать новый объект GDI, выбрать его в контекст, произвести необходимые операции рисования, а затем освободить и удалить.

В ряде случаев можно обойтись небольшим количеством предопределенных в системе Windows перьев и кистей. Их хэндлы можно получить при помощи функции getStockObject с использованием ряда определенных в модуле WINDOWS.PAS идентификаторов-констант, например

hPen:=GetStockObject(WHITE_PEN); *{белое перо}*

Тогда использование пера в некотором контексте hdc может выглядеть примерно так:

```
oldPen:=SelectObject(hdc, GetStockObject(WHITE_PEN));
 {возвращает хэндл пера, которое было выбрано до того}
{операции рисования}
SelectObject(hdc, oldPen); {выбрать старое перо}
```

Однако в большинстве случаев предопределенных объектов GDI оказывается недостаточно, поэтому программист вынужден создавать свои перья, кисти и шрифты. Так как объекты GDI не являются составляющей частью контекста устройства, то хэндл контекста устройства не используется при их создании. Это означает, во-первых, что эти объекты могут создаваться и уничтожаться один раз, например, при запуске и завершении программы. Во-вторых, объекты могут использоваться (быть выбраны) сразу в нескольких контекстах устройств.

Но это, конечно, не означает, что нельзя порождать и уничтожать объекты GDI во время одного цикла рисования между BeginPaint и EndPaint.

Перья

Для создания пера используются функции:

function CreatePen(iPenStyle, iWidth, rgbColor: integer): THandle;

function CreatePenIndirect(**const** LogPen: TLogPen): THandle;

Обе они возвращают хэндл созданного пера, отличие второй от первой состоит в том, что в качестве параметра передается запись с теми же полями, что и параметры первой функции. iPenStyle определяет стиль рисуемой линии (непрерывная, штриховые, невидимая), iWidth — толщина пера в логических единицах (если указан 0, то всегда 1 пиксел), rgbColor — цвет пера. Подробнее см. Справку по Win32 API.

Тогда работа с зеленым пером будет выглядеть следующим образом:

```
hGreenPen:=CreatePen(PS_SOLID, 1, rgb(0,$FF,0) );
oldPen:=SelectObject(hdc, hGreenPen);
 {возвращает хэндл пера, которое было выбрано до того}
{операции рисования}
```

```
SelectObject(hdc, oldPen); {выбрать старое перо}
DeleteObject(hGreenPen);
```

или, что менее читаемо,

```
oldPen:=SelectObject(hdc, CreatePen(PS_SOLID, 1, rgb(0,$FF,0) ));
 {возвращает хэндл пера, которое было выбрано до того}
{операции рисования}
DeleteObject(SelectObject(hdc, oldPen)) ;
{выбрать старое перо и удалить освободившееся}
```

Объекты, удаляемые с помощью DeleteObject, не должны быть выбранными ни в одном контексте устройства, за этим должен следить программист.

Кисти

Для создания кистей служат функции:

```
function CreateSolidBrush(rgbColor: integer): THandle;
function CreateHatchBrush(iHatchStyle, rgbColor: integer): THandle;
function CreatePatternBrush(hBitmap: THandle): THandle;
function CreateBrushIndirect(const LogBrush: TLogBrush): THandle;
```

Первая создает кисть для закрашивания цветом, вторая — для закрашивания цветной штриховкой, третья — для закрашивания произвольным битовым шаблоном (размер которого, к сожалению, не может в Windows 95 превышать 8x8 пикселов). Последняя функция может быть использована вместо любой из предыдущих. Подробнее см. Справку по Win32 API.

Будучи созданной, кисть может быть выбрана в любой контекст устройства для задания способа закрашивания областей .

Шрифты

Небольшое количество хэндлов шрифтов может быть получено с помощью функции GetStockObject. Но в большинстве случаев программисту приходится создавать объекты-шрифты "вручную". Созданный шрифт должен быть выбран в контекст устройства для использования, а затем, после освобождения из контекста, удален при помощи стандартной функции DeleteObject.

Шрифты создаются функциями CreateFont или CreateFontIndirect, обе функции возвращают хэндл созданного объекта. Эти функции отличаются лишь тем, что в первом случае в функцию передается множество параметров, а во втором — одна большая запись. Рассмотрим последнюю из них.

```
function CreateFontIndirect(const LogFont: TLogFont): THandle;
```

Запись LogFont имеет следующую структуру:

```
TLogFont = packed record
   lfHeight: Longint;
   lfWidth: Longint;
   lfEscapement: Longint;
   lfOrientation: Longint;
   lfWeight: Longint;
   lfItalic: Byte;
   lfUnderline: Byte;
   lfStrikeOut: Byte;
   lfCharSet: Byte;
   lfOutPrecision: Byte;
   lfClipPrecision: Byte;
```

```
    lfQuality: Byte;
    lfPitchAndFamily: Byte;
    lfFaceName: array[0..31] of Char;
  end;
```

Значение полей этой записи таково:

lfHeight - Высота шрифта в логических координатах предполагаемого контекста устройства. Если здесь задано положительное значение, то это высота шрифта с учетом дополнительного межстрочного интервала, если отрицательное — то это минус высота символов шрифта без учета дополнительного межстрочного интервала. Для расчета высоты шрифта, соответствующей заданному значению в типографских пунктах, следует применять формулу

```
lfHeight := - PointSize*GetDeviceCaps(hDC, LOGPIXELSY) div 72;
```

lfWidth - Ширина шрифта в пунктах. Это некоторая абстрактная величина (символы в шрифте могут иметь различную ширину), показывающая, насколько сжат или растянут шрифт по горизонтали относительно своего нормального начертания. Нормальное начертание достигается, когда lfWidth=0 или lfWidth=lfHeight.

lfEscapement - Угол в десятых долях градуса, под которым выводится строка и все символы в ней относительно оси X.

lfOrientation - В Windows NT задает угол в десятых долях градуса, под которым каждый символ строки повернут относительно горизонтали. В Windows 95 не поддерживается и должен быть равен lfEscapement.

lfWeight - Жирность шрифта. 0 или 400 - обычный, 700 - полужирный. Диапазон значений — от 0 до 1000.

lfItalic, lfUnderline, lfStrikeOut - Ненулевое значение делает шрифт наклонным, подчеркнутым и зачеркнутым соответственно.

lfFaceName - Z-строка, содержащая имя шрифта, под которым он зарегистрирован в системе, например 'Times New Roman Cyr'#0.

Остальные параметры могут быть в большинстве случаев заданы нулевыми.

При выводе текста также широко используются следующие функции:

SetBkMode - устанавливает прозрачный или непрозрачный режим вывода.

SetBkColor - устанавливает цвет фона шрифта при непрозрачном выводе.

SetTextAlign - устанавливает способ трактовки координат при выводе текста.

SetTextColor - устанавливает цвет текста.

GetTextExtentPoint - возвращает длину и высоту заданной строки в логических координатах устройства.

Действие этих функций не исчерпывается текстовым выводом. Подробнее см. Справку по Win32 API.

Растровые операции

При рисовании линий и заливке областей цвет пера или кисти может не просто копироваться в закрашиваемые пикселы поверхности рисования, а объединяться с пикселами поверхности рисования побитно в соответствии с одним из 16 заданных режимов ROP2. Режим устанавливается функцией

```
function SetROP2(hdc, iDrawMode: integer): integer;
```

возвращающей предыдущий активный режим ROP2. Прочитать текущий режим можно с помощью функции

```
function GetROP2(hdc: integer): integer;
```

Наиболее употребительные растровые операции — это R2_COPYPEN (обычное рисование, устанавливается по умолчанию) и R2_XORPEN (исключающее или, при этом то, что

нарисовано, может быть удалено повторным рисованием того же самого в том же самом месте).

Операции с битовыми образами

Битовый образ — это цифровое растровое представление изображения. Изображение представляется как двумерный массив чисел, характеризующих цвет точек растра. Практически любой графический редактор позволяет сохранить или экспортировать изображение в формат .BMP, который собственно и является форматом аппаратно-независимого (device-independent) битового образа Windows (DIB). Описание формата DIB приведено в файле BMP.TXT. Также см. Справку по разделам BITMAPFILEHEADER, BITMAPINFOHEADER.

Битовые образы (bitmap) также являются объектами GDI и могут выбираться в контекст устройства. Однако логика работы с битовыми образами отличается от того, что можно делать с перьями, кистями и шрифтами.

Две основные задачи, для которых используются битовые образы в большинстве программ — это:

- показ изображения, хранящегося в файле или ресурсе (сканированной фотографии или изображения на кнопке панели инструментов);
- организация буфера в памяти для построения изображения в нем.

Если битовый образ хранится в ресурсе, то он загружается с созданием объекта bitmap при помощи вызова функции

function LoadBitmap(hInstance: THandle; lpBitmapName: PChar): THandle;

Такие объекты требуют уничтожения с помощью DeleteObject.

При необходимости загрузить изображение из файла возникает гораздо больше проблем. Их была призвана устранить функция LoadImage, появившаяся в API Windows 95, которая позволяет загрузить битовый образ, значок или курсор из ресурса или из файла. К сожалению, эта функция не поддерживает загрузку образа из файла в Windows NT, поэтому программы Windows 95, использующие эту возможность, окажутся нерабочими под NT, что было бы крайне неприятно.

Поэтому для визуализации изображения из файла лучше использовать подход, продемонстрированный в примере LAB4.PAS. В этом случае файл BMP целиком читается в область памяти, после чего с помощью функции StretchDIBits отрисовывается в контексте устройства окна. При этом объект GDI bitmap как таковой не используется.

Для визуализации объекта - битового образа необходимо проделать ряд дополнительных действий. Дело в том, что Windows API не предоставляет средств для рисования объектов битовых образов в контексте устройства — для этого обязательно должен быть задействован контекст памяти, создаваемый функцией CreateCompatibleDC. Упрощенный вариант этих действий представлен в следующей процедуре, визуализирующей битовый образ на контексте устройства. Полный вариант процедуры представлен в тексте примера LAB31.PAS.

```
procedure DrawBitmap(hdc, hBitmap: THandle; xStart,yStart: integer);
  var bm: TBitmap;
      hdcMem: THandle;
begin
  hdcMem:=CreateCompatibleDC(hdc);
  SelectObject(hdcMem, hBitmap);
  GetObject(hBitmap, sizeof(TBitmap), @bm);
  BitBlt(hdc, xStart, yStart, bm.bmWidth, bm.bmHeight,
         hdcMem,0,0,SRCCOPY);
  DeleteDC(hdcMem);
```

```
end;
```

Вначале создается контекст памяти и в него выбирается объект - битовый образ. Выбор битового образа отождествляет контекст памяти с битовым образом в том смысле, что любая операция рисования, проводимая в контексте памяти, приводит к немедленному изменению битового образа, а изменение содержимого битового образа можно рассматривать как рисование в контексте памяти (хотя это никак не отображается на экране).

Функция GetObject позволяет получить информацию о любом объекте GDI по его хэндлу. Нас интересует, в данном случае, ширина и высота объекта - битового образа.

Наконец, выполняется пересылка прямоугольной области из контекста памяти в контекст устройства. Так как в контексте памяти был выбран битовый образ, это приводит наконец к визуализации битового образа в контексте устройства HDC. При необходимости выполнить визуализацию с растяжением или сжатием, следовало бы вместо bitBlt использовать функцию StretchBlt. Подробнее об использованных функциях см. Справку по Win32 API.

> Если стоит задача просто отобразить файл .BMP, то для этого наиболее подходит функция StretchDIBits, рисующая прямоугольную область битового образа, содержащегося в памяти в формате BMP, на контексте устройства с возможным сжатием/растяжением. При этом, если образ масштабируется, то для многоцветных образов желательно установить функцией SetStretchBltMode режим сжатия ColorOnColor. Подробнее см. Справку по Win32 API.

Для организации рисования в буфере также используется похожая технология: создается контекст памяти, создается битовый образ нужного размера с той же организацией цвета, что и в контексте устройства, после чего битовый образ выбирается в контекст памяти. Затем в контексте памяти производятся требуемые операции рисования, и получившаяся картина переносится в контекст устройства при помощи BitBlt или StretchBlt, т.е.

```
hdcMem:=CreateCompatibleDC(hdc);
hBitmap:=CreateCompatibleBitmap(hdc,width,height);
selectObject(hdcMem, hBitmap);
{рисование на hdcMem}
BitBlt(hdc, .... ,hdcMem, ....);
DeleteDC(hdcMem);
DeleteObject(hBitmap);
```

В примере LAB4.PAS такая технология реализована в процедуре PaintInBuffer.

Битовые образы могут использоваться для создания кистей с требуемым рисунком заполнения. При этом используется только та часть битового образа, которая не превышает по размерам квадрата 8x8 пикселов. Кисть на основе битового образа может быть построена, например, следующим образом:

```
hBitmap:=LoadBitmap(hInstance,'PATTERN1'); // загрузить ресурс
hBrush:=CreatePatternBrush(hBitmap); // создать кисть на основе bitmap
DeleteObject(hBitmap); // bitmap больше не нужен
hdc:=getDC(hwnd);      // получить контекст устройства
selectObject(hdc,hBrush);  // выбрать кисть
rectangle(hdc,10,10,20,20); // прямоугольник будет заполнен рисунком кисти
releaseDC(hdc);   // освободить контекст устройства
DeleteObject(hBrush);  // удалить созданную кисть
```

Системы координат устройства

Большинство функций GDI работает в логических координатах устройства. Это означает, что в общем случае координаты исчисляются не в пикселах, а в произвольных единицах, при этом начало координат не обязательно находится в верхнем левом углу окна, оси не

обязательно направлены слева-направо и сверху-вниз, масштаб по осям также может быть различным.

То, как логические координаты будут преобразовываться в физические (пиксельные) координаты устройства, определяется четырьмя парами параметров, которые обозначим как xWinExt, xWinOrg, xViewExt, xViewOrg, yWinExt, yWinOrg, yViewExt и yViewOrg. Физический смысл этих параметров следующий:

- xViewOrg, yViewOrg - положение начала координат в физической (пиксельной) сетке относительно левого верхнего угла области рисования.
- xWinOrg, yWinOrg - положение точки начала координат, заданной (xViewOrg, yViewOrg), в системе логических координат.
- xViewExt / xWinExt - масштабный коэффициент по оси X для перевода логических координат в физические, отрицательный знак означает, что ось X направлена влево.
- yViewExt / yWinExt- масштабный коэффициент по оси Y для перевода логических координат в физические, отрицательный знак означает, что ось Y направлена вверх.

Если обозначить логические координаты как (xWin, yWin), то физические координаты в пикселах относительно левого верхнего угла области рисования вычисляются по формулам:

```
xView = (xWin - xWinOrg) * xViewExt / xWinExt + xViewOrg;
yView = (yWin - yWinOrg) * yViewExt / yWinExt + yViewOrg;
```

Соответственно,

```
xWin = (xView - xViewOrg) * xWinExt / xViewExt + xWinOrg;
yWin = (yView - yViewOrg) * yWinExt / yViewExt + yWinOrg;
```

Для выполнения этих преобразований служат функции

```
function DPtoLP(DC: HDC; var Points; Count: Integer): BOOL;
function LPtoDP(DC: HDC; var Points; Count: Integer): BOOL;
```

производящие преобразование координат из физических в логические и из логических в физические соответственно в контексте DC для Count точек типа TPoint, содержащихся в массиве Points.

В зависимости от режима отображения некоторые из этих параметров заданы жестко, некоторые — можно изменять. Для изменения перечисленных параметров служат функции:

```
function SetViewportExtEx(DC: HDC; XExt, YExt: Integer; Size: PPoint): BOOL;
function SetViewportOrgEx(DC: HDC; X, Y: Integer; Point: PPoint): BOOL;
function SetWindowExtEx(DC: HDC; XExt, YExt: Integer; Size: PPoint): BOOL;
function SetWindowOrgEx(DC: HDC; X, Y: Integer; Point: PPoint): BOOL;
```

Параметры — контекст устройства, значения по осям, а также указатель на структуру типа TPoint, в которую заносятся старые значения по осям. В качестве этого указателя может быть передан NIL. Данные для Viewport задаются в пикселах, для Window — в логических единицах в зависимости от режима отображения.

Выбор режима отображения координат осуществляется функцией

```
function SetMapMode(hdc: THandle; index: integer): integer;
```

Функция возвращает предыдущий установленный режим отображения. Возможные режимы перечислены в следующей таблице:

Режим	Единицы	Напр. X	Напр. Y	WinOrg	ViewOrg	WinExt	ViewExt
MM_TEXT	пикселы	вправо	вниз	(0,0) var	(0,0) var	(1,1) const	(1,1) const
MM_LOENGLISH	0.01"	вправо	вверх	(0,0) var	(0,0) var	(?,?) const	(?,-?) const
MM_LOMETRIC	0.1 мм	вправо	вверх	(0,0) var	(0,0) var	(?,?) const	(?,-?) const
MM_HIENGLISH	0.001"	вправо	вверх	(0,0) var	(0,0) var	(?,?) const	(?,-?) const

MM_TWIPS	1/20 pt	вправо	вверх	(0,0) var	(0,0) var	(?,?) const	(?,-?) const
MM_HIMETRIC	0.01 мм	вправо	вверх	(0,0) var	(0,0) var	(?,?) const	(?,-?) const
MM_ISOTROPIC	произв. dx = dy	произв.	произв.	(0,0) var	(0,0) var	var, kx=ky	var, kx=ky
MM_ANISOTROPIC	произв	произв.	произв.	(0,0) var	(0,0) var	var	var

1" = 2.54 см

1 pt = 1/72" = 0.353 мм

Режим MM_TEXT

Этот режим удобен для работы в пиксельных координатах устройства и выбирается по умолчанию при получении контекста устройства. Ось Y направлена вниз. Шаг логических координат совпадает с шагом физических и составляет 1 пиксел по каждой оси, начало координат по умолчанию расположено в левом верхнем углу области рисования.

WinExt=ViewExt=1 в этом случае означает лишь то, что масштабирование при пересчете координат не производится; эти параметры не связаны с физической шириной и высотой области рисования.

Можно изменять положение начала координат.

Метрические режимы

К метрическим относятся режимы MM_LOENGLISH, MM_HIENGLISH, MM_LOMETRIC, MM_HIMETRIC и MM_TWIPS. Метрическими их называют потому, что логические координаты по обоим осям соответствуют единицам измерения метрической, дюймовой или типографской системы мер.

Windows устанавливает значение WinExt и ViewExt исходя из имеющихся в системе сведений о разрешающей способности устройства отображения, причем yWinExt и yViewExt имеют противоположный знак. Ось Y таким образом оказывается направленной вверх. По умолчанию начало координат находится в левом верхнем углу области рисования.

Можно изменять положение начала координат.

Режим MM_ISOTROPIC

В этом режиме имеется возможность задать не только положение начала координат, но и шаг логической координатной сетки. Особенностью данного режима является то, что Windows принудительно обеспечивает в нем равный шаг по обоим осям координат. Типичная область применения этого режима — отрисовка изображений в виртуальной системе координат, которые должны изменять размеры вместе с изменением размеров окон, но должны сохранять пропорции по горизонтали и вертикали, т.е. не сплющиваться и не вытягиваться. Прямоугольники с равными длинами сторон в таком случае выглядят квадратами, эллипсы с равными полуосями — кругами.

Например, пусть требуется, чтобы изображение с виртуальным размером 512x512 пикселов, всегда помещалось по центру окна, не изменяя соотношения сторон, и всегда занимало максимально возможную площадь. Пусть также необходимо, чтобы координатная ось Y была направлена вверх. Для этого следует воспользоваться следующей последовательностью команд:

```
SetMapMode(hdc, MM_ISOTROPIC);
SetWindowExtEx(hdc, 512, 512, nil);
SetViewportExtEx(hdc, cxClient, -cyClient, nil); {минус -> Y вверх}
SetViewportOrgEx(hdc, cxClient div 2, cyClient div 2, nil);
{Далее рисовать, как будто размер области рисования 512x512}
```

В этом случае cxClient и cyClient — это ширина и высота рабочей области в пикселах. Получить эти величины можно, например, вызвав функцию

function GetClientRect(hWnd: HWND; **var** lpRect: TRect): BOOL;
тогда
cxClient:=lpRect.right; cyClient:=lpRect.bottom;

Режим MM_ANISOTROPIC

Этот режим дает полную свободу в управлении шагом координат по осям. Он, в отличие от MM_ISOTROPIC, подходит тогда, когда нужно, чтобы изображение занимало всю площадь окна независимо от его размеров, при этом сплющиваясь и растягиваясь. Другой вариант использования этого режима — эмуляция текстового терминала с использованием моноширинного шрифта. При этом задается грубая координатная сетка, шаг которой по вертикали равен высоте символа, а по горизонтали — его ширине. После этого для вывода текста с помощью TextOut можно использовать "текстовые" координаты "столбец-строка".

Это реализуется так:

```
SetMapMode(hdc, MM_ANISOTROPIC);
SetWindowExtEx(hdc, 1, 1, nil);
SetViewportExtEx(hdc, CharWidth, CharHeight, nil);
```

где вместо CharWidth и CharHeight должны быть подставлены ширина и высота символов.

Задание

Выполняется при самостоятельной подготовке:

1. Изучить теоретическую часть описания ЛР и материал соответствующих лекций.

2. По материалам Справки (Help) изучить описание следующих функций API и связанных с ними структур данных:

BeginPaint, Endpaint, GetDC, GetWindowDC, ReleaseDC, CreateCompatibleDC, DeleteDC, GetDeviceCaps, RGB, MoveToEx, LineTo, Ellipse, Arc, Pie, Chord, Rectangle, GetStockObject, SelectObject, CreatePen, CreatePenIndirect, DeleteObject, CreateSolidBrush, CreateHatchBrush, CreatePatternBrush, CreateBrushIndirect, CreateFontIndirect, SetROP2, SetBkMode, SetBkColor, TextOut, ExtTextOut, DrawText, SetTextAlign, SetTextColor, GetTextExtentPoint, LoadBitmap, LoadImage, StretchDIBits, BitBlt, StretchBlt, DPtoLP, LPtoDP, SetMapMode, SetViewportExrEx, SetWindowExtEx, SetViewportOrgEx, SetWindowOrgEx.

Выполняется в лаборатории:

1. Включить компьютер. Запустить систему Delphi 2.0.

2. Загрузить исходный текст примера LAB4.PAS, а также модуль WINDOWS.PAS, изучить логику работы с объектами GDI, битовыми образами и логическими координатами.

3. Откомпилировать и запустить пример. Изучить поведение созданного окна.

4. Написать и отладить программу по индивидуальному заданию (см. ниже). Продемонстрировать результаты работы преподавателю.

5. Завершить работу в Delphi. Оставить компьютер включенным.

Варианты индивидуальных заданий

Программа должна создавать окно, в котором фон заполнен рисунком из файла .BMP, а в прямоугольной рамке заданной конфигурации выведен текст. Файл .BMP может быть создан с помощью стандартного редактора Paint или с помощью image Editor из комплекта по-

ставки Delphi. Текст не должен "выезжать" за и на границу рамки. Возможны произвольные дополнения визуального оформления окна по инициативе студента.

№	Заполнение рисунком	Рамка	Текст
1	растянут	10x10 см.	Строки программы, 12pt, Arial
2	растянут	70% по ширине и высоте окна	Надпись по диагонали, 12pt, Arial
3	растянут	максимальный квадрат по центру окна	Строки программы, 14pt, Courier
4	мозаичное	10x10 см.	Надпись по диагонали, 14pt, Courier
5	мозаичное	70% по ширине и высоте окна	Строки программы, 12pt, Times
6	мозаичное	максимальный квадрат по центру окна	Надпись по диагонали, 12pt, Times
7	растянут	10x10 см.	Строки программы, 14pt, Courier
8	растянут	70% по ширине и высоте окна	Надпись по диагонали, 14pt, Courier
9	растянут	максимальный квадрат по центру окна	Строки программы, 12pt, Arial
10	мозаичное	10x10 см.	Надпись по диагонали, 12pt, Arial
11	мозаичное	70% по ширине и высоте окна	Строки программы, 14pt, Times
12	мозаичное	максимальный квадрат по центру окна	Надпись по диагонали, 14pt, Times

Лабораторная работа №3

Изучение механизмов ввода информации от клавиатуры, мыши и таймера

Работа выполняется в системе программирования Borland Delphi 2.0 в среде Windows 95. Использование средств визуального программирования и классов VCL не допускается, то есть Delphi используется просто как 32-разрядный компилятор с языка Паскаль для операционной системы Windows.

Теоретические положения

Операционная система Windows извещает запущенные процессы о событиях, в том числе о событиях от устройств ввода, путем посылки им сообщений. Сообщения от клавиатуры, мыши и таймера помещаются в очереди сообщений потоков, откуда извлекаются потоками-получателями и отправляются на обработку в оконные процедуры. Логика прохождения сообщений от клавиатуры, мыши и таймера заслуживает отдельного описания.

Клавиатура

Сообщения от клавиатуры можно разделить на аппаратные и символьные.

Аппаратные сообщения возникают при нажатии и отпускании клавиш на клавиатуре и содержат аппаратно-независимый виртуальный код нажатой клавиши Windows, а также скэн-код. Аппаратные сообщения помещаются в очередь сообщений процесса операционной системой.

Символьные сообщения содержат код символа (или управляющего символа), если была нажата алфавитно-цифровая клавиша. Символ зависит не только от нажатой клавиши, но и от положения клавиш Shift, CapsLock, установленного языкового драйвера. Символьные сообщения помещаются в очередь при обработке аппаратных сообщений о нажатии клавиш процедурой TranslateMessage, которая вызывается по инициативе самого процесса в цикле обработки сообщений:

```
while GetMessage(msg,0,0,0) do begin {получить очередное сообщение}
  TranslateMessage(msg);   {Windows транслирует сообщения от клавиатуры}
  DispatchMessage(msg);    {Windows вызовет оконную процедуру}
end; {выход по wm_quit, на которое GetMessage вернет FALSE}
```

Аппаратные сообщения клавиатуры

Аппаратные сообщения от клавиатуры накапливаются в общесистемной очереди, откуда посылаются в очередь сообщений окна, имеющего фокус ввода. Пока окно не обработает сообщение, следующее сообщение не посылается. Это связано с тем, что обработка сообщения от клавиатуры может приводить к смене окна, имеющего фокус ввода, и последующие сообщения должны быть адресованы уже новому окну. Поэтому, если система по каким-либо причинам не может осуществить обработку сообщений от клавиатуры в темпе их поступления, наличие общесистемной очереди клавиатурных сообщений оказывается оправданным. (Окно извещается о получении или потере фокуса ввода посылкой ему сообщений WM_SETFOCUS и WM_KILLFOCUS.)

При нажатии клавиши окну посылается сообщение WM_KEYDOWN или WM_SYSKEYDOWN, при отпускании — WM_KEYUP или WM_SYSKEYUP. При длительном удержании клавиши нажатой сообщения о нажатии клавиши посылаются

многократно. Сообщения, в идентификаторах которых фигурирует слово "SYS", свидетельствуют о том, что клавиша нажимается при нажатой клавише Alt.

Все перечисленные аппаратные сообщения от клавиатуры в параметре WPARAM имеют виртуальный код клавиши Windows. Эти коды имеют идентификаторы, начинающиеся с VK_ , и их можно найти в файле WINDOWS.PAS.

В LPARAM содержится дополнительная информация о нажатии клавиши, структура которой такова:

Биты	Размер	Значение
0..15	16	Счетчик повторений
16..23	8	Скэн-код
24	1	Флаг принадлежности расширенной клавиатуре
29	1	Флаг контекста. "0" для обычных и "1" для системных сообщений.
30	1	Предыдущее состояние клавиши. "1" - была нажата, "0" - была отпущена.
31	1	Состояние клавиши. "1" - нажата (KEYDOWN), "0" - отпущена (KEYUP).

Счетчик повторений содержит количество повторов нажатия клавиши, которое может отличаться от 1 в случае, когда пользователь нажимает и удерживает клавишу, а программа не в состоянии обработать сообщения клавиатуры в темпе их поступления. В такой ситуации Windows не помещает в очередь отдельные сообщения, а накапливает нажатия в одном единственном сообщении, увеличивая поле счетчика в параметре LPARAM.

Скэн-код клавиши предоставляется в соответствующем поле, однако программа для идентификации клавиши должна использовать ее виртуальный код, передаваемый в WPARAM.

Флаг принадлежности к расширенной клавиатуре выставляется в единицу, если нажаты такие клавиши, как правый Alt, Ctrl или Shift, "серые" клавиши со знаками арифметических операций и Enter, NumLock, а также клавиши управления курсором, не относящиеся к клавиатуре цифрового набора.

Флаг контекста показывает, нажата ли клавиша Alt в момент посылки сообщения. Когда активное окно минимизировано, оно получает все клавиатурные сообщения как системные, и чтобы определить, нажата ли действительно клавиша Alt, используется данный флаг.

Как правило, пользовательский процесс не должен реагировать на системные сообщения клавиатуры, передавая их в DefWindowProc. Чаще всего программа должна обрабатывать только сообщения WM_KEYDOWN от клавиш управления курсором.

Символьные сообщения клавиатуры

После обработки аппаратного сообщения в TranslateMessage в очередь могут быть помещены символьные сообщения WM_CHAR, WM_SYSCHAR, WM_DEADCHAR и WM_SYSDEADCHAR. Сообщения со словом "SYS" являются следствием сообщений WM_SYSKEYDOWN, без него — WM_KEYDOWN.

Символьные сообщения содержат код нажатого символа в WPARAM, а в LPARAM содержится та же информация, что и у породившего их аппаратного сообщения, в том числе — количество повторов символа.

DEADCHAR (немой символ) имеет смысл для некоторых европейских раскладок клавиатуры, когда символ с диакритическим знаком вводится путем последовательного нажатия клавиши соответствующего знака (например — апострофа) и алфавитной клавиши. В результате окно получает два символьных сообщения: WM_DEADCHAR с символом диакритического знака и, при последующем нажатии алфавитной клавиши, WM_CHAR с символом с диакритическим знаком, например символом Ó . Если введенный после немого символа алфавитный символ не может иметь диакритического знака, окно получит не два, а три сообщения: WM_DEADCHAR и WM_CHAR с диакритическим знаком и WM_CHAR с буквой алфавита. Очевидно, что поскольку логика обработки немых символов реализована в Windows, пользовательской программе обычно нет необходимости обрабатывать сообщения о немых символах.

Состояние клавиатуры

Windows содержит ряд функций для опроса и переключения состояния клавиатуры. Эти функции возвращают не текущее состояние клавиатуры в момент их вызова, а состояние клавиатуры на момент посылки последнего извлеченного из очереди клавиатурного сообщения. Таким образом можно считать, что к каждому сообщению о нажатии клавиши "привязана" информация о состоянии всех клавиш в этот момент, и именно с этой информацией о "логическом состоянии клавиш" можно производить операции.

function GetKeyState(nVirtKey: Integer): SmallInt;

Функция GetKeyState возвращает логическое состояние клавиши с соответствующим виртуальным кодом. Старший бит 16-разрядного результата взведен, если клавиша нажата, и сброшен, если клавиша отпущена. Младший бит для клавиш управления режимом (CapsLock, NumLock, ScrollLock) показывает, активен соответствующий режим ("1") или нет ("0").

Следующие две функции позволяют прочитать и модифицировать логическое состояние клавиатуры целиком. Возвращаемое логическое значение свидетельствует об успехе или неуспехе производимой операции.

type TKeyboardState = **array**[0..255] **of** Byte;

function GetKeyboardState(**var** KeyState: TKeyboardState): BOOL;

function SetKeyboardState(**var** KeyState: TKeyboardState): BOOL;

Старший бит соответствующего виртуальному коду клавиши байта взведен, когда клавиша нажата, и сброшен, когда отпущена. Младший бит по-прежнему показывает режим для клавиш управления режимом.

Для определения физического, т.е. аппаратного состояния клавиши в текущий момент служит функция GetAsyncKeyState с параметрами, аналогичными GetKeyState.

Мышь

Использование указательного устройства в графических средах, подобных Windows, является стандартом. В качестве такового в Windows используется устройство типа "мышь" с одной, двумя или тремя кнопками. Для Windows 95 стандартной является двухкнопочная мышь.

Функция GetSystemMetrics позволяет определить следующие параметры:

getSystemMetrics(SM_MOUSEPRESENT) — не 0, если в системе установлена мышь

getSystemMetrics(SM_CMOUSEBUTTONS) — количество кнопок мыши

getSystemMetrics(SM_SWAPBUTTON) — не 0, если включено зеркальное отображение кнопок мыши (правая воспринимается как левая и наоборот).

С событиями от мыши связано 21 сообщение Windows. Если курсор мыши находится в рабочей области окна, окно получает следующие сообщения:

Движение курсора: WM_MOUSEMOVE

Нажатие кнопок:

Кнопка	**Нажатие**	**Отпускание**	**Двойное (второе) нажатие***
Левая	WM_LBUTTONDOWN	WM_LBUTTONUP	WM_LBUTTONDBLCLK
Правая	WM_RBUTTONDOWN	WM_RBUTTONUP	WM_RBUTTONDBLCLK
Средняя	WM_MBUTTONDOWN	WM_MBUTTONUP	WM_MBUTTONDBLCLK

* Сообщения о двойных щелчках окна получают лишь в том случае, когда в стиле их оконного класса указан флаг CS_DBLCLKS. В противном случае для отслеживания многократных щелчков в оконной процедуре удобно использовать функцию GetMessageTime для определения моментов посылки сообщений и интервалов между ними.

Для всех этих сообщений параметры следующие:

wParam - состояние кнопок, а также клавиш Ctrl и Shift клавиатуры; OR-комбинация констант MK_LBUTTON, MK_RBUTTON, MK_MBUTTON, MK_SHIFT и MK_CONTROL.

lParam - координаты курсора в пикселах относительно левого верхнего угла рабочей области, X=loWord(lParam); Y=hiWord(lParam);

В отличие от сообщений клавиатуры, получение сообщений от мыши определяется не активностью окна, а положением курсора мыши в области окна. Сообщения о нажатии кнопок мыши могут не сопровождаться соответствующими сообщениями об их отпускании, если курсор вдруг будет выведен за границы окна.

Windows помещает сообщения от мыши в очередь сообщений процесса только в том случае, если там нет однотипных сообщений. Из-за этого в случае перегруженности системы возможно "пропадание" некоторых действий с мышью. Например, если пользователь щелкнул кнопкой мыши в поле окна, то соответствующие сообщения помещаются в очередь, но пока они не будут извлечены оттуда программой, следующие нажатия и отпускания кнопки мыши в этом окне будут пропадать, т.е. очередные сообщения WM_xBUTTONDOWN и WM_xBUTTONUP в очередь помещены не будут.

Захват мыши

Часто окну бывает необходимо получать сообщения от мыши даже в том случае, если курсор мыши покинет пределы рабочей области. Например, если пользователь нажимает кнопку мыши для перетаскивания какого либо объекта в окне графического редактора, это окно всегда должно знать, когда кнопка мыши будет отпущена, чтобы завершить или отменить операцию перетаскивания. Для этого в обработчике сообщения WM_LBUTTONDOWN окном должен быть выполнен захват мыши, а в WM_LBUTTONUP — освобождение захвата.

Захват мыши осуществляется функцией SetCapture(hWnd), которая возвращает нулевое значение или хэндл окна, осуществлявшего захват мыши до вызова этой функции. Освобождение захвата осуществляется вызовом функции ReleaseCapture без параметров. Окно, захватившее мышь, получает сообщения от мыши, когда курсор находится вне его,

только в том случае, если кнопка мыши нажата. В противном случае сообщения мыши проходят стандартным образом, невзирая на захват.

Сообщения мыши нерабочей области

Окно получает извещение о действиях с мышью и тогда, когда курсор находится на нерабочей области окна — на заголовке или на рамке. Названия этих сообщений имеют в своем составе аббревиатуру NC (non-client), и трактовка их параметров отличается от трактовки параметров сообщений для рабочей области.

Это сообщения:

WM_NCMOUSEMOVE, WM_NCLBUTTONDOWN, WM_NCRBUTTONDOWN, WM_NCLMUTTONDOWN, WM_NCLBUTTONUP, WM_NCRBUTTONUP, WM_NCLMUTTONUP, WM_NCLBUTTONDBLCLK, WM_NCRBUTTONDBLCLK, WM_NCLMUTTONDBLCLK.

wParam - обозначает зону окна. Полный список возможных вариантов можно получить в описании сообщения WM_NCHITTEST.

lParam - как и для сообщений рабочей области — координаты курсора, но в координатной системе всего экрана, а не рабочей области окна.

Преобразование координат между клиентской и экранной системами координат осуществляется функциями

function ClientToScreen(hWnd: HWND; **var** lpPoint: TPoint): BOOL;
function ScreenToClient(hWnd: HWND; **var** lpPoint: TPoint): BOOL;

Сообщение WM_NCHITTEST

Прежде чем программа получит любое сообщение от мыши, Windows запрашивает окно о том, к какой его области относится положение курсора мыши. Для этого при возникновении каждого системного события от мыши Windows посылает соответствующему окну сообщение WM_NCHITTEST, в ответ на которое оконная процедура должна вернуть значение, определяющее, где в данном окне (в рабочей области, на рамке, на заголовке и т.д.) находится указанное Windows положение курсора. Положение курсора задается в параметре lParam так же, как и в остальных сообщениях от мыши; параметр wParam не используется. В зависимости от результатов обработки сообщения WM_NCHITTEST будут сгенерированы дальнейшие сообщения для рабочей или нерабочей области окна.

Обработка этого сообщения внутри оконной процедуры пользовательского процесса открывает ряд интересных возможностей. Например, следующий фрагмент позволяет создать окно, которое можно перетаскивать по экрану не только "ухватываясь за заголовок", но и "за рабочую область":

```
case Msg of

  .....
  wm_nchittest: begin
    result:=DefWindowProc(hwnd,msg,wparam,lparam);
      {Если попали в рабочую область, то обманываем Windows
       и говорим, что в заголовок}
    if result=HTCLIENT then result:=HTCAPTION;
  end;
  .....
end;
```

Чаще всего пользовательская программа обрабатывает:
HTCAPTION - строка заголовка окна
HTCLIENT - рабочая область
HTMENU - меню
HTNOWHERE - вне окна на свободном поле экрана
HTREDUCE - кнопка минимизации окна
HTSYSMENU - значок системного меню в левом верхнем углу окна
HTZOOM - кнопка максимизации окна

Таймер

Программа Windows может запросить операционную систему, чтобы та ставила ее в известность об истечении заданных промежутков времени. При этом операционная система создает логический таймер с заданным периодом, а программа начинает получать сообщения WM_TIMER. Следует заметить, что использование логического таймера не гарантирует точного отслеживания заданных промежутков времени; эти сообщения следует использовать для грубого задания интервалов времени в несколько секунд, а также для инициации периодического обновления отображаемой информации о выполняемом процессе, когда точность не важна, лишь бы информация менялась "примерно раз в N секунд".

Сообщения WM_TIMER имеют, подобно сообщениям WM_PAINT, низкий приоритет. Это означает, что сообщение WM_TIMER может быть извлечено из очереди только тогда, когда в очереди нет других сообщений. Если процесс не успел обработать предыдущее сообщение WM_TIMER, находящееся в очереди, то следующее сообщение WM_TIMER не ставится в очередь, и событие от таймера просто теряется.

Период логического таймера задается в миллисекундах и может лежать в интервале от 1 до $FFFFFFFF (что соответствует приблизительно 50 суткам). Однако для отсчета периода логических таймеров Windows использует аппаратный таймер компьютера, который в компьютерах IBM PC AT срабатывает 18.2 раза в секунду, т.е. с периодом приблизительно в 55 мсек. Из этого следует, во-первых, что задание периода таймера менее 55 мсек. бессмысленно, так как программа в этом случае все равно будет получать одно сообщение таймера каждые 55 мсек. Во-вторых, задаваемый период логического таймера всегда округляется вниз до ближайшего значения, соответствующего целому количеству срабатываний аппаратного таймера. Так, задав интервал в 1000 мсек., пользовательский процесс будет получать сообщения WM_TIMER каждые 989 мсек., точнее, с таким интервалом они будут попадать в его очередь сообщений, извлечь их оттуда вовремя — его собственная проблема.

Логический таймер создается и активизируется функцией SetTimer, а уничтожается (освобождается) функцией KillTimer:

```
function SetTimer(hWnd: THandle; nIDEvent, uElapse: integer; lpTimerFunc: pointer): integer;
function KillTimer(hWnd: THandle; uIDEvent: integer): BOOL;
```

Здесь hWnd - хэндл окна, создающего таймер, nIDEvent - числовой идентификатор таймера в программе, uElapse - период таймера в миллисекундах, lpTimerFunc - необязательный указатель на процедуру реакции на события от таймера (может быть nil).

SetTimer возвращает числовой идентификатор таймера или 0, если по каким-либо причинам таймер не может быть создан. В предыдущих версиях Windows количество логических таймеров в системе было ограничено (16, затем 32). В Windows 95 эти ограничения сняты, и невозможность создания таймера перестала быть обычной ситуацией.

Сообщение WM_TIMER имеет следующие параметры:

wParam - числовой идентификатор таймера.

lParam - указатель на функцию обработки, если таковая имеется.

Windows предоставляет две возможности по обработке сообщений таймера: в оконной процедуре и в специальной процедуре обработки. В первом случае в качестве lpTimerFunc в SetTimer передается nil, и сообщение WM_TIMER должно обрабатываться наравне со всеми остальными в общем операторе CASE оконной процедуры. Во втором случае в программе должна быть описана процедура со следующим набором параметров (имя — произвольное):

procedure TimerProc(hwnd:THandle; uMsg, idEvent, time: integer); **stdcall**;

Указатель на эту процедуру (@TimerProc) передается в качестве lpTimerFunc в SetTimer, и при каждом получении сообщения WM_TIMER процедура DispatchMessage будет вызывать не оконную процедуру, а процедуру TimerProc. Параметры этой процедуры — хэндл окна, тип сообщения (всегда WM_TIMER), числовой идентификатор таймера и системное время в момент получения сообщения.

Windows при использовании процедуры реакции таймера предоставляет возможность создавать таймер без окна, указав в параметре hwnd функции SetTimer значение 0. В этом случае значение параметра uIDEvent игнорируется системой, и функция возвращает номер таймера, назначенный автоматически. Этот номер должен быть впоследствии передан в KillTimer, параметр hwnd при этом также должен быть установлен в 0.

Задание

Выполняется при самостоятельной подготовке:

1. Изучить теоретическую часть описания ЛР и материал соответствующих лекций.

2. По материалам Справки (Help) изучить описание следующих сообщений Windows и связанных с ними функций и структур данных:

Сообщения: WM_KEYDOWN, WM_KEYUP, WM_CHAR, WM_DEADCHAR, WM_SYSKEYDOWN, WM_SYSKEYUP, WM_SYSCHAR, WM_NCMOUSEMOVE, WM_NCLBUTTONDOWN, WM_NCRBUTTONDOWN, WM_NCLMUTTONDOWN, WM_NCLBUTTONUP, WM_NCRBUTTONUP, WM_NCLMUTTONUP, WM_NCLBUTTONDBLCLK, WM_NCRBUTTONDBLCLK, WM_NCLMUTTONDBLCLK, WM_NCHITTEST, WM_TIMER.

Функции: TranslateMessage, GetKeyState, GetKeyboardState, SetKeyboardState, GetAsyncKeyState, Setcapture, ReleaseCapture, ClientToScreen, ScreenToClient, SetTimer, KillTimer, GetTickCount.

3. Продумать алгоритм решения индивидуального задания.

Выполняется в лаборатории:

1. Включить компьютер. Запустить систему Delphi 2.0.

2. Загрузить исходный текст примера LAB5.PAS, а также модуль WINDOWS.PAS, изучить логику работы программы.

3. Откомпилировать и запустить пример. Изучить поведение созданного окна.

4. Написать и отладить программу по индивидуальному заданию (см. ниже). Продемонстрировать результаты работы преподавателю.

5. Завершить работу в Delphi. Оставить компьютер включенным.

Варианты заданий:

1. Программа-секундомер. При нажатии клавиши Enter в рабочей области окна начинает отображаться отсчитываемое время. Точность отсчета - десятые доли секунды. При повторном нажатии Enter отсчет останавливается. Третье нажатие Enter обнуляет время. Окно секундомера — масштабируемое, без заголовка. Перемещение окна реализовать при помощи перетаскивания мышью "за клиентскую область".

Примечание: Пытаясь отслеживать время с точностью до 0.1 секунды, помните о невысокой точности сообщений таймера Windows.

2. В рабочей области окна существует единственный объект, представляющий собой геометрическую фигуру, отличную от прямоугольника (например, круг или треугольник). Нажатием цифровых клавиш 1..8 цвет объекта изменяется на один из 8 возможных. Стрелками осуществляется перемещение объекта на 1 пиксел в любую из 4 сторон. Предусмотреть также возможность перетаскивания объекта мышью.

Примечание: объект удобно представлять как регион, см. соответствующую группу функций в справке.

3. Программа - телетайп. Используя моноширинный шрифт (например, Courier) программа организует окно с размером клиентской области 80x25 символов. Алфавитно-цифровые символы выводятся на этот экран. Символ конца строки приводит к переходу в начало следующей строки, остальные управляющие символы обрабатывать не нужно. При достижении конца строки набор продолжается в следующей строке. При достижении конца экрана содержимое прокручивается на 1 строку, при этом верхняя строка теряется. При щелчке левой кнопкой мыши по какому-либо знакоместу весь экран телетайпа заполняется соответствующим символом, находящимся в указанной позиции. Окно очищается при нажатии клавиши Esc.

4. Программа - монитор состояния клавиатуры. В окне в произвольном виде отображается состояние (нажато-отпущено) клавиш на клавиатуре. Можно ограничиться только алфавитно-цифровыми клавишами. Отслеживание должно производиться даже тогда, когда окно не имеет фокуса ввода, т.е. следует асинхронно сканировать состояние клавиатуры. Окно при этом должно быть управляемым, т.е. приходящие в очередь сообщения должны обслуживаться.

5. "Уворачивающееся" окно. При попытке ввести курсор мыши в область, занятую окном (в том числе в неклиентскую область) окно изменяет свое положение таким образом, чтобы курсор мыши оказался вне окна. Окно не должно полностью уходить за границы экрана, а производимый сдвиг должен быть по возможности минимальным.

6. "Вручную", т.е. без собственной обработки WM_NCHITTEST, реализовать перетаскивание окна без заголовка "за клиентскую область". При этом желательно, чтобы в процессе перетаскивания текущее положение окна отображалось прямоугольной рамкой. Закрывать окно по нажатию клавиши Esc.

7. Прототип программы-"калькулятора", складывающей целые неотрицательные числа. В рабочей области отображается "индикатор" и 10 "клавиш" с цифрами, клавиши "Сброс(Esc)", "+" и "=". Последовательность применения калькулятора: набрать цифры первого числа (не более 8), нажать "+", набрать цифры второго числа (не более 8), нажать "=" - отобразится сумма, а калькулятор готов к приему очередного первого числа. При нажатии клавиши "Сброс" калькулятор переводится в состояние ожидания ввода первого числа, а сумма становится нулевой. Реализовать возможность ввода управляющих воздействий как с клавиатуры, так и мышью с помощью нарисованных в рабочей области клавиш.

Примечание: Удобно было бы реализовать ввод с клавиатуры, а затем при щелчке мышью по нарисованной клавише посылать окну сообщение WM_CHAR с кодом соответствующей клавиши клавиатуры. Пусть окно будет не масштабируемым.

8. Окно с изменяемым цветом фона, задаваемым в формате RGB (красный-зеленый-синий). Клавиши с буквами R, G и B ответственны за изменение каждой из составляющих цвета: если нажата клавиша Shift и "R", "G" или "B", то соответствующая составляющая цвета фона увеличивается на 1, при отпущенной клавише Shift - уменьшается. Из трех клавиш "R", "G" и "B" допускается одновременно нажимать (и удерживать) более одной, тогда должны изменяться несколько цветовых составляющих сразу. Цветовая составляющая не должна становиться меньше 0 и больше 255, т.е. выходить за диапазон BYTE. Предусмотреть вывод текущих значений составляющих цвета (допустимо использовать функцию IntToStr из модуля SysUtils в составе Delphi).

Примечание: в ответ на сообщение WM_KEYDOWN необходимо проверять состояние всех клавиш клавиатуры.

9. "Рисование" мышью. При нажатии левой кнопки мыши производится рисование точки. При движении мышью с нажатой левой кнопкой производится рисование линии по координатам, проходимым мышью. При нажатии правой кнопки производится заливка области (см. FloodFill). Цвет пера и кисти выбирается из 8 возможных при нажатии соответственно клавиш 1..8 и F1..F8.

Примечание: не требуется реализовывать запоминание рисунка и его восстановление в случае, если рабочая область окна по каким-то причинам станет недействительной.

10. Программа-"метроном". Рабочая область окна (или ее участок) должны периодически изменять цвет (например, с черного на белый и наоборот) через постоянные интервалы времени, например — каждую секунду. Допустимый интервал — от 0.5 секунды до 3 секунд. Изменение интервала с шагом в 0.1 секунду производится при нажатии клавиш "стрелка вверх" и "стрелка вниз", с шагом 0.5 сек. — при нажатии тех же клавиш в комбинации с клавишей Ctrl.

Лабораторная работа №6

Изучение средств Windows для построения пользовательского интерфейса

Работа выполняется в системе программирования Borland Delphi 2.0 в среде Windows 95. Использование средств визуального программирования и классов VCL не допускается, то есть Delphi используется просто как 32-разрядный компилятор с языка Паскаль для операционной системы Windows. Разрешается использовать функции работы со строками Delphi из модуля SysUtils.

Теоретические положения

При изучении теоретического материала полезно обратиться к описанию ЛР3: шаблон программы Windows.

Оконные органы управления (controls)

В системе Windows реализованы средства для создания современного пользовательского интерфейса, включающего такие элементы как кнопки (buttons), списки (listboxes), полосы прокрутки (scrollbars), окна редактирования (editboxes) и т.д. Все эти элементы управления в программе должны порождаться как дочерние окна окон программы - экземпляры предописанных классов. Такими классами являются (перечисляются идентификаторы, используемые в процедуре CreateWindow):

button - кнопка, переключатель (radio button), флажок (checkbox).
static - просто прямоугольная область.
scrollbar - горизонтальная или вертикальная полоса прокрутки.
edit - поле редактирования, одно- или многострочное.
listbox - список.
combobox - выпадающий список или список, комбинированный со строкой редактора.

При создании органа управления в процедуру CreateWindow, в отличие от создания обычного окна программы, должны быть переданы следующие параметры:

```
function CreateWindow(
  lpClassName: PChar;  // имя класса окна управления, малыми буквами, как представлено выше
  lpWindowName: PChar; // текст окна (надпись на кнопке и т.д.)
  dwStyle: DWORD;  // Стиль окна должен включать флаги WS_CHILD и (обычно)
                   // WS_VISIBLE + стиль органа управления
  X, Y: integer    // Положение и размеры органа управления в координатах
                   // рабочей области родительского окна
  nWidth, nHeight: Integer;  // Ширина и высота
  hWndParent: HWND; // Хэндл окна-родителя
  hMenu: HMENU;     // Идентификатор (номер) органа управления, определяется
                    // программистом
  hInstance: HINST;  // Экземпляр приложения, создающий окно
  lpParam: Pointer   // nil
): HWND;
```

Дочерние окна обрабатывают сообщения от мыши и клавиатуры. Когда пользователь производит с органом управления какие-либо действия, окну-родителю посылается сообщение WM_COMMAND, содержащее информацию о производимом действии и идентификатор (номер) окна управления, присвоенный ему при создании. Оконная процедура родительского окна должна уметь обрабатывать такие сообщения. Параметры сообщения WM_COMMAND:

loword(wParam) - идентификатор дочернего окна управления
hiword(wParam) - код уведомления (операции)
lParam - хэндл дочернего окна управления.

Родительское окно управляет дочерними, тоже посылая им сообщения. Эти сообщения зависят от класса окон управления.

Кроме того, для управления дочерними окнами часто используются функции ShowWindow, MoveWindow, IsWindowVisible, EnableWindow, IsWindowEnabled, SetWindowText, GetWindowText. Подробнее об этих функциях см. Справку по Win32 API.

Для переключения фокуса ввода между окнами управления в диалогах обычно используются клавиши Tab и Shift-Tab. Windows содержит средства для автоматизации этого действия без значительного усложнения пользовательской программы.

Дочерние окна управления автоматически уничтожаются при уничтожении родительского окна.

Цикл обработки сообщений для диалогов

Чаще всего оконными органами управления "оборудуются" окна, предназначенные исключительно для организации диалога с пользователем — ввода значений, установки параметров и т.д. Такие окна называют диалоговыми или просто диалогами. Windows предоставляет программисту возможность без особых затрат труда организовать переключение органов управления при нажатии пользователем стандартных комбинаций клавиш (самые распространенные комбинации — Tab и Shift-Tab). Для этого цикл обработки сообщений программы должен быть модифицирован следующим образом (это может быть как основной цикл программы, так и специально организуемый для конкретного диалога цикл обработки сообщений).

Пусть хэндл диалогового окна хранится в переменной hDlg. тогда цикл обработки сообщений должен выглядеть как:

```
while GetMessage(msg,0,0,0)=true do begin {получить очередное сообщение}
  if not IsDialogMessage(hDlg,msg)
         {Если Windows не распознает и не обрабатывает клавиатурные сообщения
          как команды переключения между оконными органами управления,
          тогда чообщение идет на стандартную обработку}
  then begin
    TranslateMessage(msg);   {Windows транслирует сообщения от клавиатуры}
    DispatchMessage(msg);    {Windows вызовет оконную процедуру}
  end;
end; {выход по wm_quit, на которое GetMessage вернет FALSE}
```

Функция IsDialogMessage в этом случае анализирует текущее сообщение MSG и, если сообщение содержит информацию о нажатии стандартной комбинации клавиш, служащей для перемещения между органами управления диалога, обрабатывает его. Если сообщение обрабатывается, функция возвращает TRUE и программа обрабатывать такое сообщение уже не должна, в противном случае возвращается FALSE и сообщение обрабатывается на общих основаниях.

В ходе выполнения функции IsDialogMessage Windows проверяет, есть ли в окне с указанным хэндлом (hDlg) оконные органы управления вообще, и если они есть, то выполняется последовательная посылка ряда сообщений им для смены текущего органа управления.

Кнопки (класс button)

Поддерживаются следующие стили кнопок:

Стиль	Характеристика

BS_PUSHBUTTON	Обычная кнопка ("нажимаемая кнопка").
BS_DEFPUSHBUTTON	Кнопка с более жирной рамкой. В диалогах, описываемых в ресурсах, считается нажатой при нажатии пользователем клавиши Enter.
BS_CHECKBOX	Флажок, т.е. надпись, слева или справа от которой расположено квадратное поле, которое может быть отмечено крестиком.
BS_AUTOCHECKBOX	То же, что предыдущее, но отметка или снятие отметки происходит автоматически, не требуя участия родительского окна.
BS_3STATE	То же, что флажок, но возможно также "третье состояние", когда поле флажка отображается серым цветом.
BS_AUTO3STATE	То же, что предыдущее, но переключение между тремя возможными состояниями происходит автоматически при щелчке по флажку.
BS_RADIOBUTTON	Индикатор исключающего выбора: кружок, слева от которого размещена строка текста. Кружок может быть отмечен точкой в центре.
BS_AUTORADIOBUTTON	То же, что предыдущее, но при щелчке по индикатору он автоматически становится выбранным, а все другие индикаторы в той же группе (того же родительского окна) — невыбранными.
BS_GROUPBOX	Рамка с текстом в верхнем левом углу, обычно служит для визуального объединения радио-кнопок в группу. Само по себе не реагирует на события от клавиатуры и мыши.
BS_OWNERDRAW	Кнопка пользователя, отрисовкой внешнего вида кнопки должна заниматься оконная процедура родительского окна, обрабатывая сообщение WM_DRAWITEM.

При нажатии кнопок в сообщении WM_COMMAND содержится код уведомления BN_CLICKED.

Родительское окно может посылать окнам кнопок управляющие сообщения:

Сообщение	**Описание**
BM_GETCHECK	Возвращает для флажков и индикаторов выбора состояние выбора: 0 (не помечено), 1 (помечено), 2 (третье состояние).
BM_SETCHECK	Устанавливает состояние выбора для флажков и индикаторов выбора. В wParam должен содержаться код состояния (см. выше).
BM_GETSTATE	Возвращает состояние как комбинацию битовых флагов: биты 0 и 1 — содержат код, возвращаемый BM_GETCHECK; бит 2 — нажата (1) или отпущена (0) кнопка; бит 3 — имеет ли кнопка фокус.
BM_SETSTATE	Для нажимаемых кнопок устанавливает состояние. wParam = 0 — кнопка отпущена, wParam = 1 — кнопка нажата.
BM_SETSTYLE	Позволяет изменить стиль кнопки после ее создания. wParam - содержать новый стиль как комбинация битовых флагов,

lParam - 1 - перерисовать кнопку, 0 - не перерисовывать.

Например, изменить состояние окна-флажка с хэндлом hCheckBox1 на противоположное можно с помощью следующей конструкции:

```
SendMessage(hCheckBox1, BM_SETCHECK,
            SendMessage(hCheckBox1, BM_GETCHECK,0,0) xor 1,
            0);
```

Для того, чтобы изменить способ отображения кнопок или их реакцию на внешние события, необходимо создать собственный оконный класс на основе класса button (рассматривается ниже).

Статические окна (класс static)

Эти окна отображаются как закрашенные прямоугольники или рамки с текстом. Обычно эти окна не обрабатывают событий от клавиатуры и мыши. Они могут использоваться как контейнеры для других дочерних окон управления. Подробнее см. Справку по Win32 API, функция CreateWindow.

Полосы прокрутки

Окна Windows могут иметь оконные горизонтальные и вертикальные полосы прокрутки, что определяется включением в стиль окна флагов WS_VSCROLL и WS_HSCROLL. Помимо этого существуют также дочерние окна управления класса scrollbar, которые выглядят точно так же, но, в отличие от полос прокрутки окна, могут располагаться в любом месте и иметь произвольный размер. Далее будет рассмотрено использование обоих типов полос прокрутки.

Окна с полосами прокрутки (полосы — не дочерние окна)

Чтобы окно имело вертикальную полосу прокрутки, в его стиле при вызове функции CreateWindow должен присутствовать флаг WS_VSCROLL, горизонтальную — WS_HSCROLL. Такие полосы прокрутки (назовем их оконными) размещаются в окне вдоль правой и нижней границ и имеют протяженность вдоль всей границы окна в пределах рабочей области. Пространство, занятое оконными полосами прокрутки, не включается в рабочую область. Ширина полос прокрутки является общесистемным параметром и ее можно узнать с помощью функции GetSystemMetrics (см. также SystemParametesInfo).

Оконные полосы прокрутки управляются при помощи мыши, они не реагируют на нажатия клавиш, поэтому программист должен "вручную" реализовывать прокрутку при нажатии на клавиши управления курсором в оконной процедуре.

Полосы прокрутки характеризуются диапазоном, определяемым двумя целыми числами — минимальным и максимальным положением бегунка. Положение бегунка всегда дискретно и соответствует целому числу из диапазона возможных положений от минимального до максимального. Так, если диапазон задан как "от 0 до 100" (по умолчанию), то бегунок может находиться в одном из 101 возможных положений, и его положение будет возвращаться как целое число от 0 до 100. Диапазон по умолчанию не всегда удобен, поэтому есть возможность переопределить его при помощи вызова функции

```
function SetScrollRange(hWnd: HWND; nBar, nMinPos, nMaxPos: Integer;
                        bRedraw: BOOL): BOOL;
```

Узнать текущий диапазон оконной полосы прокрутки можно с помощью функции

```
function GetScrollRange(hWnd: HWND; nBar: Integer; var nMinPos, nMaxPos:
                        Integer): BOOL;
```

Здесь hWnd - хэндл окна, nBar - SB_VERT или SB_HORZ, nMinPos и nMaxPos - минимальное и максимальное положение полос прокрутки, bRedraw - определяет, требуется ли перерисовка полосы прокрутки сразу после установления новых параметров. При неуспешном завершении операции возвращается FALSE.

Если не выполняется условие nMinPos<nMaxPos, то полоса прокрутки становится невидимой.

Для работы с положением бегунка используются функции

```
function SetScrollPos(hWnd: HWND; nBar, nPos: Integer; bRedraw: BOOL): Integer;
function GetScrollPos(hWnd: HWND; nBar: Integer): Integer;
```

Здесь hWnd — хэндл окна, nBar - SB_VERT или SB_HORZ, bRedraw - определяет, требуется ли перерисовка полосы прокрутки сразу после установления новых параметров. Возвращаемое значение — старое положение бегунка (оно же во втором случае — текущее).

Сообщения полос прокрутки

При действиях с полосами прокрутки оконной процедуре посылаются сообщения WM_VSCROLL (вертикальная прокрутка) и WM_HSCROLL (горизонтальная прокрутка). В младшем слове wParam содержится код операции с полосой прокрутки, lParam для сообщений оконных полос прокрутки можно игнорировать. Кодами операции в младшем слове wParam могут быть:

SB_BOTTOM	Прокрутить в самый конец
SB_ENDSCROLL	Кнопка мыши отпущена, операция завершена (это сообщение как правило можно игнорировать)
SB_LINEDOWN	Прокрутка вниз на одну строку, возникает при нажатии на стрелку внизу (справа) полосы прокрутки.
SB_LINEUP	Прокрутка вверх на одну строку, возникает при нажатии на стрелку вверху (слева) полосы прокрутки.
SB_PAGEDOWN	Прокрутка на страницу вниз, возникает при щелчке мыши на полосе прокрутки ниже (справа от) бегунка.
SB_PAGEUP	Прокрутка на страницу вверх, возникает при щелчке мыши на полосе прокрутки выше (слева от) бегунка.
SB_THUMBPOSITION	Установить позицию бегунка. Позиция содержится в старшем слове wParam. Возникает после перетаскивания бегунка мышью.
SB_THUMBTRACK	Позиция в время перетаскивания. Позиция содержится в старшем слове wParam. Возникает во время перетаскивания ползунка мышью.
SB_TOP	Прокрутка в самое начало.

Как видно из приведенной таблицы, положение бегунка в сообщениях SB_THUMBPOSITION и SB_THUMBTRACK передается в виде 16-разрядного целого, поэтому "полнофункциональная" полоса прокрутки должна иметь диапазон, определяемый числами в диапазоне от 0 до 65535. Функции SetScrollRange, SetScrollPos, GetScrollRange и GetScrollPos оперируютс 32-разрядными целыми, однако положение бегунка, выходящее за 16-разрядный диапазон, не может быть корректно передано в теле сообщения. При обработке сообщения SB_THUMBPOSITION есть возможность узнать 32-разрядное

положение бегунка при помощи функции GetScrollPos, однако в случае SB_THUMBTRACK с ее помощью узнать положение бегунка невозможно, так как функция GetScrollPos не отражает положения бегунка в процессе его перетаскивания.

При перетаскивании бегунка в очередь сообщений окна помещается множество сообщений с признаком SB_THUMBTRACK, после чего по окончании операции посылается сообщение с SB_THUMBPOSITION.

Простейший способ реакции на сообщения WM_VSCROLL и WM_HSCROLL — вызов функции ScrollWindow для прокрутки клиентской области окна. Прокрутка состоит в том, что изображение в окне соответствующим образом сдвигается, а "открывшиеся" после сдвига участки становятся недействительными, и в очередь сообщений помещается сообщение WM_PAINT. Дочерние окна и начало координат окна также сдвигаются.

Полосы прокрутки — окна класса scrollbar

Полоса прокрутки может быть помещена в произвольном месте окна. Для этого необходимо создать ее как дочернее окно управления класса scrollbar, указав стиль SBS_VERT или SBS_HORZ (см. Справку по WIN32 API: CreateWindow). Полоса прокрутки при этом занимает весь указанный в параметрах CreateWindow (или MoveWindow) прямоугольник, т.е. ее ширина и высота могут быть произвольными. Системные настройки ширины стандартных оконных полос прокрутки можно узнать, вызвав функции GetSystemMetrix(SM_CYHSCROLL) или GetSystemMetrix(SM_CXVSCROLL).

Окна класса scrollbar, как и стандартные полосы прокрутки окна, посылают родительскому окну сообщения WM_VSCROLL или WM_HSCROLL вместо сообщений WM_COMMAND, посылаемых остальными оконными органами управления. Отличия этих сообщений от сообщений, посылаемых оконными полосами прокрутки, состоят в том, что для оконных полос прокрутки lParam содержит 0, а для дочерних окон - полос прокрутки — хэндл такого дочернего окна. В отличие от стандартных оконных полос прокрутки, окна класса scrollbar могут получать фокус и обрабатывать нажатия клавиш клавиатуры. Нажатия клавиш интерпретируются следующим образом:

Клавиша	wParam сообщения VM_xSCROLL
Home	SB_TOP
End	SB_BOTTOM
Page Up	SB_PAGEUP
PageDown	SB_PAGEDOWN
Стрека влево, стрелка вверх	SB_LINEUP
Стрелка вправо, стрелка вниз	SB_LINEDOWN

Управление полосами прокрутки класса scrollbar осуществляется теми же функциями GetScrollPos, GetScrollRange, SetScrollPos и SetScrollRange, что и для стандартных оконных полос прокрутки, однако в параметре hWnd следует указывать хэндл самой полосы прокрутки, а в nBar — константу SB_CTL.

Окна редактирования (класс edit)

Окна редактирования представляют собой одно- или многострочные текстовые редакторы. Они могут иметь клавиатурный фокус и позволяют вводить текст, редактировать его при помощи клавиш управления, выделять при помощи клавиш и мыши и производить

операции копирования и вставки с буфером обмена при помощи клавиш Ctrl-Ins, Shift-Ins и Shift-Del. Окна редактирования могут иметь собственные стандартные полосы прокрутки, что определяется включением флагов WS_VSCROLL и WS_HSCROLL в стиль окна редактирования при его создании.

Окно редактирования создается как экземпляр класса edit. В стиле окна могут присутствовать следующие специфичные для окон-редакторов флаги:

ES_MULTILINE Многострочный редактор.

ES_AUTOHSCROLL Автоматически используется горизонтальная полоса прокрутки, если текст не умещается по ширине окна. При отсутствии этого флага производится автоматический перенос на следующую строку.

ES_AUTOVSCROLL Автоматически появляется вертикальная полоса прокрутки, если текст не умещается по высоте окна.

ES_NOHIDESEL Выделенный текст подсвечивается, даже когда окно теряет фокус, иначе при потере фокуса подсветка текста гасится.

С помощью функции SetWindowText можно задать текст в окне редактирования, с помощью GetWindowtextLength и GetWindowText — получить текст. Многострочные окна редактирования воспринимают последовательность #13#10 как конец строки; следующие символы выводятся со следующей строки, а символы конца строки не отображаются. Окна редактирования не обрабатывают символ табуляции (#9) и отображают его в виде вертикальной черты ("|"). Чтобы в обычном окне (не в диалоге) обрабатывать нажатия клавиш Tab, Shift-Tab, Enter как управляющие, необходимо вводить новую оконную процедуру для окон редактирования (см. ниже). Размер редактируемого текста ограничен примерно 32 килобайтами, при этом используется область памяти, выделенная приложению, содержащему окно редактора.

Окна редактора посылают родительскому окну сообщения WM_COMMAND со следующими кодами уведомления в старшем слове wParam:

EN_SETFOCUS Редактор получил фокус ввода

EN_KILLFOCUS Редактор потерял фокус ввода

EN_CHANGE Содержимое изменилось, посылается после вывода на экран

EN_UPDATE Содержимое изменилось, посылается перед обновлением экрана

EN_ERRSPACE Буфер редактирования переполнился

EN_MAXTEXT Буфер редактирования переполнился при вставке

Программа может управлять окном редактора, посылая ему сообщения. Наиболее важные из них:

EM_GETLINECOUNT — возвращает число строк в многострочном редакторе.

EM_LINEINDEX, wParam = номер строки — возвращает смещение начала строки относительно начала буфера редактирования. Строки нумеруются с нуля. wParam=-1 позволяет узнать смещение строки, содержащей курсор.

EM_LINELENGTH, wParam = номер строки — возвращает длину указанной строки.

EM_GETLINE, wParam = номер строки, lParam = указатель на буфер — копирует указанную строку в заданный буфер, заканчивая ее нулем.

См. также сообщения WM_CUT, WM_COPY, WM_PASTE, EM_GETSEL, EM_SETSEL, EM_REPLACESEL, с помощью которых реализуются операции с буфером обмена.

Списки (класс listbox)

Окна класса listbox представляют собой списки Windows. Список — это набор текстовых строк, которые выводятся в прямоугольном прокручиваемом окне в один или несколько столбцов. Окно списка может получать фокус и способно обрабатывать сообщения клавиатуры. Когда окно списка имеет фокус, одна из строк отображается с пунктирной рамкой и считается имеющей фокус. Кроме того, строки могут быть выделены цветом, в этом случае они считаются выбранными. Наличие у строки фокуса в общем случае не означает, что она выбрана.

По умолчанию списки не посылают родительскому окну сообщения WM_COMMAND; чтобы посылка извещений происходила, необходимо включать в стиль окна списка флаг LBS_NOTIFY. Другими важными флагами стиля являются:

LBS_SORT	Строки списка автоматически сортируются
LBS_MULTIPLESEL	Позволяет выделять несколько строк списка (список с множественным выбором)

Для того, чтобы список имел рамку и полосу прокрутки, необходимо включить в стиль окна списка флаги WS_BORDER и WS_VSCROLL. Интересные результаты получаются при использовании с окном списка флагов WS_CAPTION, WS_SIZEBOX, WS_HSCROLL.

Для управления списком используются следующие сообщения, посылаемые окну списка при помощи функции SendMessage:

LB_ADDSTRING, lParam = указатель на Z-строку — добавляет строку в конец несортированного списка, в сортированном списке строка оказывается на нужном месте в соответствии с алгоритмом сортировки.

LB_INSERTSTRING, wParam = позиция, lParam = указатель на Z-строку — вставляет строку в указанной позиции, все остальные строки сдвигаются. Если список сортированный, пересортировка не производится. При wParam=-1 строка добавляется в конец списка.

LB_DELETSTRING, wParam = позиция — удаляет строку в заданной позиции.

LB_RESETCONTENT — полная очистка списка.

WM_SETREDRAW, wParam = 0 - разрешить, 1 - запретить — устанавливает флаг, разрешающий или запрещающий перерисовку списка при изменении его содержимого. При последовательном добавлении или удалении нескольких строк полезно сбросить этот флаг перед началом операции и взвести по окончании.

LB_GETCOUNT — возвращает количество элементов в списке.

LB_SETCURSEL, wParam = номер строки — в списке с одиночным выбором делает указанную строку выбранной, при wParam=-1 ни одна строка не выбрана.

LB_SELECTSTRING, wParam = номер первой строки, lParam = указатель на Z-строку для поиска — делает выбранной первую подходящую строку, начиная с wParam. Если wParam=-1, то поиск начинается с начала списка. Выбирается строка, начинающаяся со строки, задаваемой lParam. При невозможности найти подходящую строку возвращается LB_ERR.

LB_GETCURSEL — возвращает номер текущей выбранной строки или -1.

LB_GETTEXTLEN, wParam = номер строки — возвращает длину указанной строки.

LB_GETTEXT, wParam = номер строки, lParam = адрес буфера — записывает в буфер заданную строку списка, завершая ее нулем.

LB_SETSEL, wParam = 0 - снять выделение, 1 - включить выделение, lParam - номер строки — в списках с множественным выбором выделяет или снимает выделение указанного элемента списка, если lParam=-1, то выделяется или снимается выделение у всех элементов списка.

LB_GETSEL, wParam = номер строки — возвращает 0, если строка не выбрана, и другое значение, если выбрана.

Окна списков в свою очередь извещают родительские окна о действиях с ними путем посылки им сообщений WM_COMMAND со следующими кодами уведомления в старшем слове wParam:

LBN_ERRSPACE	При добавлении строки не хватило памяти.
LBN_SELCHANGE	Изменен текущий выбор.
LBN_DBLCLICK	Был двойной щелчок мышью по списку.
LBN_SETFOCUS	Список получил фокус.
LBN_KILLFOCUS	Список потерял фокус.

Создание собственных окон управления. Subclassing и superclassing

Windows позволяет программисту создавать собственные оконные классы на основе уже существующих и модифицировать поведение уже созданных окон. В Справке по Win 32 API (из Software Development Kit) при описании этих вопросов используются термины subclassing (создание подклассов) и superclassing (создание суперклассов). Эти термины представляются не очень удачными, поскольку в объектно-ориентированном программировании термин "суперкласс" имеют несколько иное значение, а здесь наблюдается прямая аналогия с принципами ООП. "Subclassing" в документации по Windows означает подмену оконной процедуры для конкретного окна или для всего существующего класса другой оконной процедурой, "superclassing" — создание нового оконного класса на основе уже существующего с заменой оконной процедуры. (В ООП суперклассом принято называть класс-предок, в документации же по Windows под суперклассом подразумевается надстройка над существующим классом, т.е. то, что в ООП называлось бы классом-потомком.)

Замена оконной процедуры существующего окна

Этот прием используется, когда необходимо модифицировать поведение существующего окна. Используя его можно, например, сделать так, чтобы некоторые клавиатурные сообщения от дочернего окна управления, имеющего фокус, попадали в родительское окно (например, о нажатиях клавиш Tab и Enter). Для этого необходимо:

1. Описать новую оконную процедуру, предусмотрев в ней новую обработку для нужных сообщений; остальные сообщения должны передаваться старой оконной процедуре.

2. Прочитать адрес заменяемой оконной процедуры из локальной памяти окна.

3. Записать адрес новой оконной процедуры в локальную память окна.

Новая оконная процедура может поступать с сообщениями следующим образом:

- передавать их на обработку старой оконной процедуре;

- модифицировать сообщения и передавать их старой процедуре;

- обрабатывать сообщения самостоятельно, не передавая старой процедуре.

Вместо вызова DefWindowProc в новой оконной процедуре сообщения как правило передаются старой оконной процедуре при помощи функции

```
function CallWindowProc(lpPrevWndFunc: TFNWndProc;
        hWnd: HWND; Msg: UINT; wParam: WPARAM; lParam: LPARAM): LRESULT;
```

Первый параметр этой функции — указатель на старую оконную процедуру, остальные параметры — параметры для вызова оконной процедуры.

Для получения адреса оконной процедуры окна hwnd используется вызов

```
pOldProc := pointer( GetWindowLong(hwnd, GWL_WNDPROC) );
```

получаемое 32-битное целое должно быть затем преобразовано к типу pointer. Можно совместить получение адреса старой оконной процедуры с заменой его на новый, так как функция SetWindowLong возвращает старое значение изменяемой ячейки памяти окна:

```
pOldProc := pointer( SetWindowLong(hwnd, GWL_WNDPROC, integer(@NewWndProc) ) );
```

Создание нового оконного класса на основе существующего

Часто бывает необходимо изменить поведение не одного окна, а всех окон какого-либо класса, например, чтобы все кнопки передавали родительскому окну на обработку сообщения от клавиатуры. Для этого можно изменить оконную процедуру у каждой из создаваемых кнопок, но логичнее было бы создать новый класс кнопок, которые ведут себя так же, как и кнопки button, но еще и отдают сообщения от клавиатуры главному окну.

Для этого необходимо:

1. Получить описание базового класса.

2. Модифицировать описание базового класса, создав таким образом описание нового класса. В частности, обычно вводится новая оконная процедура, обрабатывающая часть сообщений и передающая остальные старой процедуре при помощи вызова CallWindowProc, как это описано выше.

3. Зарегистрировать новый класс в системе.

4. Использовать зарегистрированный класс для порождения окон.

Описание класса в виде структуры TWndClassEx можно получить с помощью функции GetClassInfoEx, затем в полученной структуре заменяются поля указателя на оконную процедуру и имени класса. Указатель на оконную процедуру базового класса должен быть сохранен для ее вызова из оконной процедуры нового класса. Модифицированное описание класса регистрируется с помощью RegisterClassEx.

Задание

Выполняется при самостоятельной подготовке:

1. Изучить теоретическую часть описания ЛР и материал соответствующих лекций.

2. По материалам Справки (Help) изучить описание следующих сообщений Windows и связанных с ними функций и структур данных:

Сообщения:

WM_COMMAND, WM_VSCROLL, WM_HSCROLL;

BM_GETCHECK, BM_SETCHECK, BM_GETSTATE, BM_SETSTATE, BM_SETSTYLE;

EM_GETLINECOUNT, EM_LINEINDEX, EM_LINELENGTH, EM_GETLINE, WM_CUT, WM_COPY, WM_PASTE, EM_GETSEL, EM_SETSEL, EM_REPLACESEL;

LB_RESETCONTENT , LB_ADDSTRING, LB_INSERTSTRING, LB_DELETSTRING, WM_SETREDRAW, LB_GETCOUNT, LB_SETCURSEL, LB_SELECTSTRING, LB_GETCURSEL, LB_GETTEXTLEN, LB_GETTEXT, LB_SETSEL, LB_GETSEL.

Функции:

SetWindowText, GetWindowtextLength, GetWindowText, SendMessage, CreateWindow, ShowWindow, MoveWindow, ScrollWindow, IsWindowVisible, EnableWindow, IsWindowEnabled, GetWindowLong, GetDlgCtrlId, SetScrollPos, GetScrollPos, SetScrollRange, GetScrollRange, GetWindowLong, SetWindowLong, SendDlgItemMessage, CheckDlgButton, CheckRadioButton, IsDlgButtonChecked.

3. Продумать алгоритм решения индивидуального задания.

Выполняется в лаборатории:

1. Включить компьютер. Запустить систему Delphi 2.0.

2. Загрузить исходный текст примера LAB6.PAS, а также модули WINDOWS.PAS, MESSAGES.PAS, изучить логику работы программы.

3. Откомпилировать и запустить пример. Изучить поведение созданных окон.

4. Написать и отладить программу по индивидуальному заданию (см. ниже). Продемонстрировать результаты работы преподавателю.

5. Завершить работу в Delphi. Оставить компьютер включенным.

Варианты заданий:

Написанная программа должна обеспечивать перемещение между органами управления с помощью стандартных комбинаций клавиш клавиатуры. Приветствуется расширение функциональных возможностей окна по инициативе студента, но с при выполнении основного условия задания. Разрешается использовать функции работы со строками Delphi из библиотеки SysUtils.

1. "Регулятор цвета". Окно, содержащее три полосы прокрутки и три информационных (не модифицируемых) поля ввода для красной, зеленой и синей составляющей цвета. При перемещении полосы прокрутки значение соответствующей цветовой составляющей должно отображаться в соответствующем информационном окошке. Заданный цвет должен произвольным образом отображаться в рабочей области главного окна. Каждая составляющая — число от 0 до 255.

2. "Регулятор цвета". Окно, содержащее три информационных (не модифицируемых) полосы прокрутки и три поля ввода для красной, зеленой и синей составляющей цвета. После ввода составляющей положение соответствующей полосы прокрутки должно изменяться. При неверном вводе значения должно выдаваться сообщение об ошибке. Заданный цвет должен произвольным образом отображаться в рабочей области главного окна. Каждая составляющая — число от 0 до 255.

3. "Калькулятор" с числовыми кнопками, полем индикации, клавишами четырех арифметических действий, стирания и равенства. Все клавиши должны быть реализованы в виде органов управления - кнопок.

4. "Список с поиском". Окно содержит список (изначально чем-то заполненный) и поле ввода. При выборе элемента списка в поле ввода должна появляться выбранная строка. В процессе набора в поле ввода подсветка в списке должна перемещаться к строке, начало которой совпадает с вводимой строкой. если совпадения нет, подсветка должна исчезать.

5. "Ханойская башня" или что-то отдаленно ее напоминающее. Три списка, строку из конца каждого списка можно переносить в конец любого другого списка. Проверок строк на допустимость переноса выполнять не требуется. Предусмотреть органы управления, позволяющие задать, в какой список (и из какого, если это требуется) осуществляется перенос.

6. "Транслитератор". В многострочном редакторе пользователь имеет возможность набрать произвольный русский текст ограниченной длины (2000 символов). После нажатия соответствующей кнопки текст переносится в другой редактор, при этом все русские буквы заменяются на эквивалентные латинские буквы или буквосочетания, например В→V, Ж→ZH и т.д. Редактор с транслитерированным текстом также допускает редактирование.

7. "Транслитератор". В многострочном редакторе пользователь имеет возможность набрать произвольный русский текст ограниченной длины (2000 символов). В процессе ввода (при вводе и удалении символов) текст подвергается транслитерации, при этом все русские буквы заменяются на эквивалентные латинские буквы или буквосочетания, например В→V, Ж→ZH и т.д. Транслитерированный текст отображается в другом поле ввода, редактирование в котором запрещено.

8. В программе два окна — "администратор" и "диалог". В окне администратора пользователь имеет список паролей, в который может добавлять пароль через строку редактора или удалять его (предусмотреть как). В окне диалога пароль вводится в "секретном" режиме (символы заменяются звездочками), после нажатия кнопки "Проверка" пользователю выдается сообщение (например, "Доступ открыт" и "Доступ запрещен"), свидетельствующее о наличии или отсутствии введенного пароля в списке паролей.

9. "Калькулятор" для перевода целых чисел из одной системы счисления в другую. В поле ввода набирается число в заданной системе счисления. В поле вывода появляется преобразованное число. Поддерживать системы счисления с основанием 2, 8, 10, 16. Предусмотреть орган управления для задания системы счисления.

10. "Создатель окон". Пользователь вводит параметры окна, которое он хочет создать, нажимает соответствующую кнопку, и окно с заданными параметрами появляется на экране. Числовые атрибуты вводить в редакторах, остальные выбирать из списков возможных значений. Предусмотреть как минимум ввод характеристик:

> ширина-высота, положение на экране, наличие заголовка, сист. меню, кнопок минимизации и максимизации, типа рамки, ввод строки заголовка, цвет клиентской области (хотя бы черный-белый-серый).

Лабораторная работа №7

Использование ресурсов и DLL

Работа выполняется в системе программирования Borland Delphi 2.0 в среде Windows 95 с использованием средств пакета Borland Resource Workshop, а также утилиты Image Editor и компилятора ресурсов из состава Delphi. Использование средств визуального программирования и классов VCL не допускается, то есть Delphi используется просто как 32-разрядный компилятор с языка Паскаль для операционной системы Windows.

Теоретические положения

Форматы исполнимых файлов Windows (.EXE и .DLL) подразумевают возможность хранения в теле этих файлов различных данных. Чаще всего это небольшие битовые образы, используемые в качестве пиктограмм, значков и курсоров мыши, а также текстовые строки. Эти данные располагаются не в области данных программы, не загружаются автоматически в память при загрузке модуля и недоступны через идентификаторы переменных внутри программы. Для доступа к этим данным их необходимо специальным образом загрузить из тела исполнимого файла, как если бы они располагались в дисковых файлах.

Для описания таких данных используются файлы ресурсов. Файлы ресурсов обычно имеют расширение .RES (в Delphi – еще и .DCR) и присоединяются к телу исполнимого файла на этапе компоновки, одновременно с файлами .OBJ, а также .LIB (для языка C) или .DCU (для Delphi).

Файлы ресурсов — это бинарные файлы, имеющие определенный формат; для создания и редактирования ресурсов используется компилятор ресурсов (например в Delphi — brc32.exe), позволяющий на основе текстового описания ресурсов (обычно это фалы с расширением .RC) построить бинарный файл .RES. Кроме того, существуют утилиты, позволяющие редактировать стандартные типы ресурсов непосредственно в файле .RES, а также осуществлять декомпиляцию ресурсов в текстовое описание. В этой работе для редактирования ресурсов предлагается использовать пакет Borland Resource Workshop.

Стандартными типами ресурсов являются:

- битовые образы (bitmaps)
- значки (icons)
- курсоры мыши (cursors)
- символьные строки (strings)
- меню (menus)
- комбинации "горячих клавиш" (keyboard accelerators)
- диалоги (dialog boxes)

Кроме того, можно создать ресурсы произвольного формата (ресурсы пользователя).

Подключение файлов ресурсов к программе

Для включения ресурсов в программу на Delphi файл ресурсов должен быть в явном виде упомянут в программе с использованием директивы компилятора $RESOURCE ($R), например так:

```
{$R myres.res}
```

К одной программе может быть подключено любое количество файлов ресурсов. При работе с Delphi не следует называть файл ресурсов именем проекта (главного файла), так как

такой ресурс Delphi создает автоматически, и созданный программистом ресурс будет просто стерт.

В модулях (units) языка Паскаль при компиляции модулей имена необходимых файлов ресурсов просто заносятся в тело файлов .DCU; поэтому при компоновке файлов .EXE или .DLL все файлы .RES, используемые в модулях, должны быть доступны компилятору. Ресурсы включаются в исполнимый файл только на этапе его компоновки, они не присутствуют в модулях .DCU.

Компиляция ресурсов

Компиляция текстового описания ресурсов в файл .RES осуществляется из командной строки компилятором ресурсов BRC32 (Borland Resource Compiler). Вызов (маршрут к директории расположения BRC32 должен быть прописан в строке PATH или указан в команде)

brc32 -r myres.rc

приведет к компиляции описания ресурсов myres.rc и созданию файла ресурсов myres.res.

Параметр **-r** означает, что не требуется производить немедленное включение файла ресурсов myres.res в исполнимый модуль. С опциями компилятора ресурсов можно ознакомиться, запустив brc32 без параметров.

Утилиты для работы с ресурсами

Утилита Image Editor, входящая в комплект поставки Borland Delphi, позволяет создавать и манипулировать простейшими бинарными ресурсами: битовыми образами, значками и курсорами мыши. Кроме того, она позволяет создавать и редактировать битовые образы, значки и курсоры непосредственно в теле файла с расширением .RES.

Утилита Resource Workshop кроме того позволяет работать как с бинарными (.RES), так и с текстовыми (.RC) файлами ресурсов, а также с ресурсами в теле исполнимого модуля, осуществлять их компиляцию/декомпиляцию, а также работать с ресурсами-меню, диалогами и таблицами строк.

Справка по Resource Workshop содержит также описание синтаксиса языка описания ресурсов, применяемого в файлах .RC.

Описание и использование ресурсов

Курсоры, значки и битовые образы

Битовый образ — это изображение в формате BMP. Можно использовать изображения любого размера и глубины цвета. Обычно изображения создаются в графическом редакторе и сохраняются в формате .BMP.

Значок (icon) — это растровое изображение размером 32x32 или 16x16 пикселов, монохромное или 16-цветное (в современных версиях Windows поддерживается и большая глубина цвета). В дополнение к возможным 2 или 16 цветам, пикселы значка могут быть прозрачными и инвертирующими. Прозрачные пикселы сохраняют цвет пикселов изображения, поверх которого выводится значок, инвертирующие — инвертируют. Значки хранятся в файлах с расширением .ICO. Гораздо меньшее количество графических редакторов поддерживают этот формат, в частности его поддерживают Image Editor в Delphi и Resource Workshop.

Курсор мыши — это монохромное (в современных версиях Windows допускается цветное) растровое изображение размером 32x32 пиксела. Каждый пиксел курсора может быть черным, белым, прозрачным или инвертирующим. Так как курсор предназначен для отображения текущего положения указателя мыши, в нем обязательно задается так называемая вершина (hotspot) в виде пары координат относительно левого верхнего угла битового образа курсора. При отображении положения мыши вершина курсора совмещается с текущими координатами мыши. Обычно курсоры в виде стрелки имеют вершину на конце стрелки, курсоры в виде перекрестия — в центре перекрестия и т.д. Курсоры хранятся в виде файлов с расширением .CUR. Обычно для их создания требуются специализированные графические редакторы (например Image Editor в Delphi или Resource Workshop).

При использовании компилятора ресурсов каждый значок, битовый образ или курсор обычно описываются в виде ссылки на соответствующий файл ресурса в формате

<идентификатор> <ТИП> <файл>

например:

```
picture1    BITMAP    pic1.bmp
hand        CURSOR    hand.cur
appicon     ICON      icon1.ico
```

Здесь picture1, hand и appicon — идентификаторы битового образа, курсора мыши и значка соответственно. Вместо строковых идентификаторов при описании ресурсов могут использоваться целые положительные 16-разрядные числа. Кроме того, описание битового образа ресурса может располагаться не в отдельном файле, а в файле .RC в виде шестнадцатеричного дампа памяти (такая ситуация возникает обычно в результате декомпиляции файлов .RES).

Для использования ресурсов в программе их необходимо загрузить.

Битовые образы загружаются с помощью функции

function LoadBitmap(hInstance: HINST; lpBitmapName: PAnsiChar): HBITMAP;

Функция возвращает хэндл битового образа или 0 при неудаче. В качестве первого параметра указывается хэндл экземпляра приложения, из исполнимого файла которого производится загрузка ресурса. Указав 0 в качестве hInstance можно получить доступ к некоторым стандартным битовым образам Windows (стрелки полос прокрутки и т.п., см. Справку по Win32 API). Второй параметр должен содержать либо указатель на строковой идентификатор битового образа, либо его числовой идентификатор, если в файле ресурса образ описывалась с номером. Windows определяет, передан указатель на строку или числовой идентификатор, проверяя старшее слово переданного указателя: если передано 16-битное число, то старшее слово нулевое, в то время как указатель на область данных программы всегда имеет ненулевое старшее слово.

После использования каждый загруженный с помощью LoadBitmap битовый образ должен быть уничтожен с помощью функции DeleteObject.

Для загрузки значков используется функция

function LoadIcon(hInstance: HINST; lpIconName: PChar): HICON;

Ее использование аналогично функции LoadBitmap. В отличие от функции LoadBitmap, если указанный значок уже был ранее загружен, вторая его копия не будет создаваться и будет возвращен хэндл уже загруженного значка. Загруженные с помощью LoadIcon значки не нуждаются в уничтожении.

Загруженный значок может использоваться при описании классов окон. Также имеется возможность отобразить значок в контексте устройства с помощью функции

DrawIcon(hdc,x,y,hIcon);

Курсоры загружаются при помощи функции

function LoadCursor(hInstance: HINST; lpCursorName: PAnsiChar): HCURSOR;

Она аналогична функции LoadIcon. Хэндл курсора может использоваться при описании классов окон. Кроме того, при обработке сообщения WM_MOUSEMOVE имеется возможность анализировать положение мыши и менять форму курсора при помощи функции

SetCursor(hCursor);

Если эту функцию не вызывать, то будет использован курсор, определенный в оконном классе (точнее – при любом движении мыши в окне Windows устанавливает курсор в соответствии с описанием оконного класса, поэтому если предполагается "вручную" менять форму курсора в оконной процедуре, то в классе окна в качестве хэндла курсора лучше указывать 0).

Пусть имеется файл описания ресурсов, содержащий описание:

```
123    BITMAP    pic1.bmp
ic1    ICON      icon1.ico
```

Тогда соответствующие битовый образ и значок должны быть загружены при помощи вызовов:

```
hbmp := LoadBitmap(hInstance, pointer(123) );
hicon := LoadIcon(hInstance, 'ic1');
..................... операции ..............
DeleteObject(hbmp);
```

Идентификаторы должны быть уникальными для однотипных ресурсов. Это означает, что среди всех файлов ресурсов программы не должно быть, например, двух битовых образов с идентификатором 'pic1', хотя наличие битового образа и значка с одинаковым идентификатором допускается. Тем не менее, рекомендуется всегда использовать уникальные идентификаторы ресурсов.

Таблицы строк символов

Таблицы строк символов являются также распространенным типом ресурсов. Использование этого вида ресурсов позволяет существенно облегчить процесс локализации программного обеспечения (например, перевод с английского на русский язык). Если все приглашения и сообщения, которые программа выводит на экран, хранятся в файлах ресурсов, то для переводчика исчезает необходимость работать с исходным текстом программы — достаточно лишь перевести собранные в одном месте строки. Однако отлаживать программу, все текстовые строки которой находятся в отдельном файле ресурсов, не очень удобно.

Символьные строки организуются в таблицы и объявляются с использованием ключевого слова STRINGTABLE следующим образом:

```
STRINGTABLE
{
<id1>, "<string1>"
.....................
}
```

Каждая строка в таблице записывается на отдельной строке и заключается в двойные кавычки (как в языке C), перед каждой строкой, отделенное запятой, указывается число-идентификатор. Например, может быть объявлена следующая таблица строк:

```
STRINGTABLE
{
1, "Ошибка времени выполнения"
2, "Недопустимая операция"
10, "Нормальное завершение"
}
```

Использование литеральных констант неудобно: необходимо поддерживать соответствие номеров строк в программе и в файле ресурсов. В языке C эта проблема решается за счет создания файла заголовка с расширением .H, в котором определяются числовые константы, и который подключается к файлу .RC и исходному тексту программы директивой #include. При работе с Delphi применяется такой же подход: константы, идентифицирующие символьные строки, могут быть описаны в отдельном файле в соответствии с синтаксисом языка Паскаль в виде секции **const,** и затем включены в исходный текст программы с помощью директивы компилятора $INCLUDE ($I), а в файл .RC — с помощью #include. Можно также объявить модуль (unit) языка Паскаль, интерфейсная часть которого состоит из объявлений констант, и подключить его к программе в разделе **uses**, а к описанию ресурсов — с помощью #include.

Тогда могут быть созданы файлы (Resource Workshop "умеет" делать это автоматически при работе со строками, но он, к сожалению, не всегда корректно работает с русским алфавитом):

```
——— константы ресурсов: RESCONST.PAS ———
unit resconst;
interface
const
  IDS_Runerror = 1;
  IDS_InvalidOp = 2;
  IDS_Normalterm = 10;
implementation end.
———— описание ресурсов ————————
#include "resconst.pas"
STRINGTABLE
{
IDS_RUNERROR, "Ошибка времени выполнения"
IDS_INVALIDOP, "Недопустимая операция"
IDS_NORMALTERM, "Нормальное завершение"
}
————————————————————————————
```

Строки загружаются с помощью функции LoadString в готовый буфер:

function LoadString(hInst: HINST; uID: UINT; lpBuffer: PChar; nBufferMax: Integer): Integer;

Здесь hInst — хэндл экземпляра приложения, из исполнимого файла которого загружается строка, uID — целочисленный идентификатор строки, lpBuffer — указатель на буфер для приема данных, nBufferMax — размер буфера. Если загружаемая строка превышает размер буфера, она будет усечена. Функция возвращает количество прочитанных символов. Загружаемая строка, даже усеченная, всегда заканчивается нулевым символом.

Ресурсы пользователя

Имеется возможность сохранить в виде ресурса и подключить к исполняемому модулю произвольные данные. Для этого в файле описания ресурсов должна быть ссылка на файл, содержащий подключаемые данные, например:

```
data1       MYDATA    example1.dat
```

В качестве идентификатора и типа данных может быть указана произвольная строка. Загрузить такие данные можно при помощи вызова

hData := LoadResource (hInstance, FindResource(hInstance, 'MYDATA', 'data1'));

Для работы с самими данными необходим не хэндл, а указатель, что выглядит примерно так:

pData := LockResource(hData);
{действия с pData}

Вызов функции UnlockResource, использовавшейся в Windows 3.x, в Windows 95 не требуется. Когда ресурс больше не нужен, его следует освободить при помощи

FreeResource(hData);

Подробнее о функциях LoadResource, FindResource, LockResource и FreeResource см. в Справке по Win32 API.

Меню

Простейшим способом включения меню в интерфейс Windows-программы является описание его в файле ресурсов и последующая его загрузка оттуда. Существует также множество функций API, позволяющих создавать и изменять меню в процессе работы программы, однако большинство из них здесь рассматриваться не будут.

Меню, как и дочерние окна управления, при различных действиях пользователя посылает программе сообщения WM_COMMAND.

Меню бывают двух видов: главное меню (main menu), отображаемое в виде строки или нескольких строк вверху окна, подменю (drop-down menu), выглядящие как вертикальные списки строк и появляющиеся при выборе пунктов в главном меню, а также всплывающие меню (popup menu) появляющиеся (новый стандарт пользовательского интерфейса) при щелчке правой кнопкой мыши по тому или иному органу управления в окне.

Пункты меню могут быть "разрешены" (enabled), "запрещены" (disabled) или "недоступны" (grayed — серые). При выборе пользователем разрешенных пунктов меню происходят какие-то действия (открывается всплывающее меню или программе посылается сообщение WM_COMMAND). Запрещенные и недоступные пункты меню также можно выбрать, но никаких действий при этом не происходит. Запрещенные пункты меню, в отличие от недоступных, не отображаются серым цветом и выглядят так же, как разрешенные.

Кроме того, пункты всплывающего меню могут быть "помечены" (checked), при этом слева от соответствующего пункта меню отображается значок "галочки".

Описание меню в файле ресурсов производится согласно следующему шаблону:

```
<имя_меню> MENU
{
[<список элементов меню>]
}
```

<имя_меню> — это строка или число, служащее для идентификации меню в программе при описании класса окна. Список элементов меню состоит из описаний пунктов меню или всплывающих меню.

Пункт меню (которому не соответствует подменю) описывается как

```
MENUITEM "<текст>", <идентификатор> [, <признаки>]
```

Если пункту меню соответствует подменю, то оно описывается как

```
POPUP "<текст>" [, <признаки>]
{
[<список элементов меню>]
}
```

<Текст>, заключенный в кавычки — это строка, которая отображается в описываемом пункте меню. В текст может быть включен знак "&"; символ текста, следующий за амперсантом, будет отображаться подчеркнутым и при нажатии его на клавиатуре совместно с клавишей Alt будет выбираться этот пункт меню.

<идентификатор> — это число, передаваемое в программу в параметре сообщения WM_COMMAND при выборе этого пункта меню. Удобно использовать внешние определения идентификаторов, как это было описано выше при обсуждении строковых таблиц, и подключать их к описанию ресурсов директивой #include. Имена констант для идентификаторов меню принято начинать с символов "IDM_", например IDM_EDIT.

В качестве признаков элементов главного меню используются следующие флаги, которые можно объединять с помощью символа дизъюнкции ("ИЛИ") языка C ("|" — "трубопровод"):

GRAYED - пункт недоступен и выводится серым.
INACTIVE - пункт неактивен (не генерирует WM_COMMAND), но отображается обычным образом, несовместимо с GRAYED.
MENUBREAK - следующие пункты меню отображаются с новой строки.
HELP - этот и все следующие пункты меню "прижимаются" к правому краю строки меню.

В описании всплывающего меню могут также встречаться как подменю, так и пункты меню, описываемые точно так же с помощью директив POPUP и MENUITEM. Кроме того, внутри всплывающих меню можно нарисовать горизонтальную черту, включив в качестве описания строки меню строку

```
MENUITEM SEPARATOR
```

Может дополнительно использоваться признак CHECKED, означающий, что пункт меню выбран и отмечен галочкой.

Использование меню

Для указания на использование меню из ресурса в описании оконного класса достаточно указать его имя, как оно описано в ресурсе, например

```
wndclass.lpszMenuName := 'MyMenu';
```

или, если при описании меню использовался числовой идентификатор, например, 12:

```
wndclass.lpszMenuName := pointer(12);
```

или

wndclass.lpszMenuName := '#12';

Все окна этого класса будут иметь указанное меню. Чтобы указать свое меню для конкретного окна при его создании, необходимо загрузить описание меню из ресурса:

hMenu := LoadMenu(hInstance, 'MyMenu');

и затем передать полученный хэндл меню hMenu в качестве девятого параметра функции CreateWindow.

Наконец, имея хэндл загруженного описания меню, можно изменить меню уже существующего окна, вызвав функцию

SetMenu(hwnd, hMenu);

а просто узнать хэндл меню окна — с помощью функции

GetMenu(hWnd);

Любое связанное с окном меню уничтожается при уничтожении окна. Однако загруженные, но не связанные с окнами меню должны быть явно удалены при помощи DestroyMenu(hMenu).

Основные сообщения меню

При выборе разрешенного пункта меню (щелчке по пункту меню левой кнопкой мыши или нажатии клавиши Enter) программе посылается сообщение WM_COMMAND с параметрами

loWord(wParam) - числовой идентификатор пункта меню
hiWord(wParam) = 0; lParam = 0

При выборе пункта системного меню (открываемого при щелчке по значку в левом верхнем углу окна) посылается сообщение WM_SYSCOMMAND с теми же параметрами. Для анализа команды необходимо замаскировать младшие 4 бита младшего слова wParam, т.е. выражение (loword(wParam) and $FFF0) может принимать значения sc_size, sc_move, sc_minimize, sc_maximize, sc_close и др. Сообщение WM_SYSMENU обычно передается в DefWindowProc.

При каждом перемещении указателя по меню при помощи мыши или клавиш управления курсором программе посылается сообщение WM_MENUSELECT. Оно полезно, например, для вывода подсказки по выбранному пункту меню в строке состояния окна. Параметры:

loword(wParam) - числовой идентификатор пункта меню или хэндл всплывающего меню
hiword(wParam) - флаги выбора
lParam - хэндл меню, содержащего выбранный пункт

Флаги выбора содержат логическую сумму констант MF_GRAYED, MF_DISABLED, MF_CHECKED, MF_POPUP, MF_SYSMENU, MF_MOUSESELECT и др.

Перед автоматическим выводом на экран всплывающего меню программа получает сообщение WM_INITMENUPOPUP, позволяющее привести содержимое меню в надлежащий вид, например — разрешить или запретить некоторые его пункты. Параметры:

wParam - хэндл всплывающего меню
loword(lParam) - индекс всплывающего меню
hiword(lParam) - 1 для системного меню и 0 для обычного

Для простейшего управления меню используются вызовы

```
EnableMenuItem(hMenu, id, MF_GARAYED);
EnableMenuItem(hMenu, id, MF_ENABLED);
CheckMenuItem(hMenu, id, MF_UNCHECKED);
CheckMenuItem(hMenu, id, MF_CHECKED);
```

Для управления меню (в том числе системным меню окна) и создания меню также используются следующие функции, с описанием которых можно ознакомиться в Справке по Win32 API:

CreateMenu, CreatePopupMenu, AppendMenu, GetSystemMenu, DeleteMenu, RemoveMenu, InsertMenu, ModifyMenu, DrawMenuBar, GetSubMenu, GetMenuItemCount, GetMenuItemID, TrackPopupMenu.

Таблица акселераторов

Таблицы акселераторов используются для упрощения построения интерфейса клавиатуры. В этой таблице задаются комбинации клавиш (как правило, соответствующие пунктам меню, хотя это необязательно), нажатие которых автоматически приводит к генерации сообщений WM_COMMAND с тем или иным кодом.

В файле описания ресурсов таблицы акселераторов определяются следующим образом:

```
<имя_таблицы> ACCELERATORS
{
[определения комбинаций клавиш]
}
```

В качестве имени таблицы можно указывать символьный или числовой идентификатор, как и для меню. Определения клавиш — это строки одного из четырех возможных форматов:

```
"<символ>", <числовой_идентификатор> [, SHIFT] [, CONTROL] [, ALT]
"^<символ>", <числовой_идентификатор> [, SHIFT] [, CONTROL] [, ALT]
<ASCII-код>,  <числовой_идентификатор>, ASCII [, SHIFT] [, CONTROL] [, ALT]
<виртуальный_код>, <числовой_идентификатор>, VIRTKEY [, SHIFT] [, CONTROL] [,
ALT]
```

Числовой идентификатор быстрой клавиши передается в параметрах сообщения WM_COMMAND. Слова SHIFT, CONTROL и ALT означают, что указанные регистровые клавиши образуют комбинацию с клавишей, код которой указывается в первом параметре.

Первый способ определения клавиши подразумевает, что в первом параметре указан символ, соответствующий алфавитно-цифровой клавише, второй — что это комбинация символа с клавишей Ctrl (фраза CONTROL в этом случае ничего не меняет), в третьем случае первым параметром должен быть числовой код буквы, в последнем случае — виртуальный код клавиши Windows. Наиболее употребительными являются первый и четвертый варианты.

Использование таблицы акселераторов

Для использования описанных в файле ресурсов комбинаций клавиш их необходимо загрузить, а также несколько модифицировать цикл выборки сообщений.

Загрузка таблицы акселераторов и получение ее хэндла осуществляется с помощью функции LoadAccelerators:

hAccel := LoadAccelerators(hInstance, 'MyAccel');

Цикл обработки сообщений дополняется вызовом функции проверки комбинаций клавиш по таблице:

```
while getMessage(msg, 0,0,0) do
  if not TranslateAccelerator(hwnd, hAccel, msg) then begin
    TranslateMessage(msg);
    DispatchMessage(msg);
  end;
```

Функция TranslateAccelerator проверяет, является ли сообщение msg сообщением клавиатуры и, если это так, проводит поиск комбинации нажатых клавиш в таблице акселераторов. Если комбинация найдена, вызывается оконная процедура окна hwnd и ей передается сообщение WM_COMMAND с кодом, идентифицирующим нажатую комбинацию, функция при этом возвращает TRUE. Для сообщений, не приводящих к посылке сообщения WM_COMMAND при трансляции, функция возвращает FALSE.

Параметр hwnd функции TranslateAccelerator позволяет сосредоточить обработку акселераторов в одной оконной процедуре: какое бы окно не являлось получателем сообщения msg (хэндл окна получателя содержится в теле сообщения), сообщение WM_COMMAND будет передано в оконную процедуру принудительно указанного окна hwnd. В качестве hwnd программа обычно должна передавать либо хэндл главного окна, либо хэндл ее активного окна.

Сообщение WM_COMMAND, порождаемое при нажатии быстрой клавиши, имеет следующие параметры:

loword(wParam) - числовой идентификатор комбинации клавиш из таблицы акселераторов

hiword(wparam) = 1; lParam = 0;

Если акселератор соответствует тому или иному пункту меню (имеет тот же идентификатор), то помимо сообщения WM_COMMAND той же оконной процедуре посылаются сообщения активизации меню, т.е. WM_INITMENU, WM_INITMENUPOPUP, WM_MENUSELECT. Если клавиша соответствует запрещенному пункту меню, то сообщения WM_COMMAND и прочие не посылаются.

Окна диалога

В файле ресурса могут быть также описаны диалоговые окна программы. Диалоговые окна как правило предназначены для ввода пользователем данных, которые невозможно или неудобно вводить при помощи меню. Диалоговые окна содержат дочерние окна управления (кнопки, поля ввода текста, радио-кнопки и кнопки-флажки), в описании вида и положения которых и состоит описание диалога.

При описании диалога требуется задавать множество координат, и это неудобно делать вручную, а еще неудобнее редактировать такое описание. Кроме того, координаты в диалогах, описываемых в ресурсах, задаются не в физических пикселах, а в особой координатной сетке (с шагом 1/8 высоты шрифта диалога по вертикали и 1/4 ширины шрифта по горизонтали), из-за чего диалоги при разных разрешениях экрана выглядят примерно одинаково по размерам. Для облегчения процесса описания диалогов (как, впрочем, и меню) созданы различные инструменты, в частности такая функция есть в пакете Borland Resource Work-

shop. С его помощью можно "нарисовать" внешний вид окна диалога и затем сохранить в виде текстового описания ресурса или непосредственно в бинарном файле ресурса.

Для обеспечения функционирования диалога необходимо описать так называемую процедуру диалога, которая очень похожа на оконную процедуру. Принципиальное отличие между диалоговой процедурой и обычной оконной процедурой состоит в том, что ее вызов производится из настоящей оконной процедуры окна диалога, которая находится "в недрах" Windows; и диалоговой процедуре передаются не все сообщения, которые получает оконная процедура. Заголовок диалоговой процедуры должен быть следующим (имя процедуры — любое):

```
function DlgProc(hDlg: HWND; Msg: UINT;
                 wParam: WPARAM; lParam: LPARAM): longint; stdcall;
```

Если функция обрабатывает сообщение, она должна возвращать ненулевое значение, иначе — 0. Сообщения, которые не обрабатываются, не должны передаваться ни в DefWindowProc, ни в DefDlgProc, так как их обработка и так сосредоточена в оконной процедуре диалога. Вызов DefDlgProc внутри диалоговой процедуры приводит к рекурсии (оттуда снова будет вызвана диалоговая процедура), в результате чего приложение намертво "подвешивает" операционную систему.

Процедура диалога обычно обрабатывает как минимум два сообщения: WM_INITDIALOG, которое посылается диалоговому окну при его инициализации, и WM_COMMAND — сообщения от дочерних окон управления.

Если в ответ на WM_INITDIALOG процедура диалога возвращает TRUE, то Windows дает фокус первому из элементов управления диалога (его хэндл передается в wParam); при инициализации диалога можно изначально дать фокус другому элементу управления с помощью функции SetFocus, при этом обработчик WM_INITDIALOG должен вернуть FALSE.

При попытке использовать SetFocus возникает проблема: так как диалоговое окно создается Windows, то в программе неизвестны хэндлы дочерних органов управления. Однако эту информацию можно легко получить, зная числовой идентификатор органа управления, задаваемый в файле ресурса, при помощи функции GetDlgItem:

```
function GetDlgItem(hDlg: THandle; id: integer): THandle;
```

Сообщения WM_COMMAND от дочерних окон управления и способы управления ими достаточно подробно рассмотрены в теоретическом материале к предыдущей работе.

Для посылки сообщений органам управления в API реализована функция

```
function SendDlgItemMessage(hDlg: HWND; nIDDlgItem: Integer;
                 Msg: UINT; wParam: WPARAM; lParam: LPARAM): Longint;
```

работающая аналогично SendMessage, при этом получателем сообщения является дочернее окно диалога hDlg с идентификатором nIDDlgItem (при этом не требуется знать хэндл этого органа управления).

В окнах диалога Windows автоматически поддерживает интерфейс клавиатуры, позволяющий перемещать фокус от одного дочернего окна управления к другому при помощи клавиши Tab и клавиш управления курсором.

Диалог загружается и активизируется при помощи функции DialogBox. Возврат из этой функции не производится, пока в ответ на какое-либо событие в диалоговой процедуре не будет вызвана функция EndDialog, т.е. функция DialogBox активизирует модальный диалог.

function DialogBox(hInstance: HINST; lpTemplate: PChar;
hWndParent: HWND; lpDialogFunc: pointer): Integer;

lpTemplate — Z-строка, содержащая имя ресурса диалога, hWndParent — хэндл окна-собственника диалога (обычно — главного окна программы, хотя можно указать 0), lpDialog-Function — указатель на процедуру диалога. Возвращаемое значение определяется программистом при вызове EndDialog внутри процедуры диалога.

function EndDialog(hDlg: HWND; nResult: Integer): BOOL;

hDlg — хэндл окна диалога (передается в процедуру диалога в качестве параметра), nResult — код завершения диалога, возвращаемый в вызывающую диалог программу. Определены некоторые стандартные коды, например ID_OK, ID_CANCEL, ID_ABORT, ID_IGNORE, ID_YES, ID_NO, ID_CLOSE. Диалог должен завершаться явным вызовом EndDialog в ответ на нажатие некоторой кнопки (обычно это Ok и Cancel), а также при получении диалоговой процедурой сообщения WM_CLOSE.

Существует возможность работать с немодальными диалоговыми окнами, используя для их создания функцию CreateDialog, однако этот вопрос здесь рассматриваться не будет.

Библиотеки DLL

DLL (Dynamic-Link Library, динамически связываемая библиотека) – это файл, содержащий набор готовых к исполнению процедур, а также, возможно, ресурсы. Функции DLL могут вызываться исполняемыми модулями (.EXE) или функциями других DLL, причем одновременно к функциям DLL может обращаться неограниченное количество других модулей.

Термин "динамическое связывание" означает, что функции библиотеки не включаются в код использующей ее программы, а вызываются из отдельно существующей библиотеки в процессе работы программы.

Механизм DLL является основой Windows: все функции API собраны в DLL-библиотеках, и исключительно благодаря поддержке механизма DLL программы способны общаться с операционной системой вообще. Написание собственной DLL и размещение ее в общедоступном каталоге можно с полным основанием считать расширением функций ОС.

Подключение DLL может осуществляться как автоматически на этапе загрузки программы (load-time linking, статическая загрузка), так и "вручную" во время работы программы посредством ряда функций API (runtime linking, динамическая загрузка). Код библиотеки загружается в память один раз, сколько бы процессов ею ни пользовались, и выгружается, когда все использующие процессы завершаются. Принудительное аварийное завершение процесса, загрузившего DLL, часто приводит к тому, что код DLL не выгружается и бесполезно занимает память.

DLL могут быть написаны на разных языках с использованием различных моделей вызова. В документации к DLL обязательно должен быть отражен момент, связанный с моделью вызова функций (правилами передачи параметров). Наиболее часто используемые модели вызова, поддерживаемые большинством компиляторов – это модели, обозначаемые в Delphi как **StdCall** (модель вызова функций WinAPI), **CDecl** (модель языка C) и **Pascal** (модель языка Паскаль).

Синтаксис исходного текста DLL

Исходный текст библиотеки DLL на языке Borland Pascal 8, используемом Delphi, чем-то напоминает исходный текст модуля (Unit) и выглядит следующим образом:

```
library <имя>;  // вместо Program или Unit
uses <обычный список используемых модулей>;

  <Объявление процедур и функций>

exports
  Proc1 index 10,
  Proc2 name 'ProcedureTwo',
  Proc3,
  .....;


begin
  <возможные начальные установки, напр. - для глоб. переменных>
end.
```

Стандартным образом, как для исполнимого модуля .EXE, в теле программы или в unit'ах описываются процедуры и данные.

Во фразе **exports** через запятую перечисляются идентификаторы экспортируемых, т.е. предназначенных для вызова внешними программами, процедур. Фраза **exports** может встречаться в тексте сколько угодно раз и перемежаться с объявлением процедур, единственное условие – процедура должна сначала объявляться, а затем экспортироваться. Рекомендуется объявлять экспортируемые процедуры с моделью вызова **StdCall**, гарантированно поддерживаемой трансляторами всех языков программирования для Windows. Экспортировать можно и процедуры из модулей.

Для каждой экспортируемой процедуры может быть в явном виде указан числовой идентификатор (компилятор по умолчанию присваивает последовательные числа) и символьный идентификатор (по умолчанию берется идентификатор из программы и переводится в верхний регистр). Именно по этим идентификаторам использующие DLL программы обращаются к ее функциям. В документации Microsoft рекомендуется пользоваться исключительно символьными идентификаторами, хотя это несколько замедляет процесс загрузки DLL.

Глобальные переменные DLL, в отличие от модуля (Unit), не могут быть экспортированы. Кроме того, если DLL используется в нескольких процессах, то для каждого процесса создается отдельная копия глобальных переменных DLL, и поэтому глобальные переменные DLL в принципе не могут быть использованы для передачи данных между процессами даже в случае доступа к ним через функции самой DLL.

DLL может содержать ресурсы, для их подключения используется все та же директива компилятора {$R}. Так как DLL является законченным исполнимым файлом, в отличие от модуля Unit, то ресурсы физически включаются в тело DLL.

Поиск используемых DLL

При использовании DLL может быть указан маршрут к ее файлу. Если маршрут указан, то файл должен находиться в указанной директории, иначе он не будет найден. Если же маршрут не указан (только имя файла), то поиск производится в следующем порядке:

1. Директория, откуда была загружена использующая DLL программа.
2. Текущая директория.
3. Директория SYSTEM32 операционной системы Windows 98/NT.
4. Директория SYSTEM операционной системы Windows.
5. Директория, куда установлены сами Windows.

6. Директории, перечисленные в переменной окружения PATH (обычно настраивается в файле autoexec.bat).

Подключение DLL на этапе загрузки

Для автоматического подключения функций DLL они объявляются в использующей программе как внешние при помощи директивы **external**. Параметры, естественно, должны быть описаны тоже. Например, если рассматриваемая в предыдущем разделе DLL скомпилирована в TESTLIB.DLL, а три процедуры, упомянутые в **export** имеют соответственно один, два и три целых 32-разрядных параметра и модель вызова **StdCall**, то они могут импортироваться следующим образом:

```
const testlib='testlib.dll';
procedure FirstProc(a:integer); external testlib index 10; stdcall;
procedure SecondProc(a,b:integer); external testlib name 'ProcedureTwo'; stdcall;
procedure PROC3(a,b,c:integer); external testlib; stdcall;
```

Третья процедура в этом случае должна называться точно так же, как и в библиотеке, так как для связывания будет использовано имя 'PROC3'.

После такого объявления и загрузки программы будет автоматически загружена библиотека TESTLIB.DLL, и при вызове в программе функций FirstProc, SecondProc и PROC3 будут вызываться соответствующие функции DLL.

Часто при создании DLL удобно сразу описать отдельный интерфейсный модуль (Unit) для импорта ее функций, а затем просто подключать такой модуль к использующей программе в фразе **uses**. Примером такого интерфейсного модуля является модуль WINDOWS, используемый во всех программах Delphi для доступа к функциям API.

При невозможности подключить библиотеку на этапе загрузки использующая ее программа не запускается.

Подключение DLL на этапе выполнения

Такой способ подключения обычно используется в двух случаях:

- когда функции DLL используются только в течение определенного этапа работы программы (например – аутентификация пользователя при запуске);

- когда программе требуются ресурсы из DLL.

Для подключения DLL используются функции:

function LoadLibrary(lpLibFileName: PChar): HMODULE;

function LoadLibraryEx(lpLibFileName: PChar; hFile: THandle; dwFlags: DWORD): HMODULE;

В качестве первого параметра передается указатель на строку с именем файла исполнимого модуля. С помощью этих функций можно загружать как DLL, так и EXE-файлы. Параметр hFile второй функции не используется и должен быть равен нулю. В параметре dwFlags можно задать дополнительные параметры загрузки библиотеки, самый интересный из которых – флаг LOAD_LIBRARY_AS_DATAFILE, полезный при работе с ресурсами библиотеки. Возвращаемое значение – хэндл загруженного исполнимого модуля, применяемый для функций API, или 0 при неудачной загрузке.

После того, как библиотека больше не требуется, использующая программа должна освободить ее при помощи вызова функции

function FreeLibrary(module: hModule):boolean;

В качестве параметра передается хэндл модуля, полученный при загрузке библиотеки.

Для доступа к функциям динамически загружаемых DLL необходимо получать адрес точки входа каждой функции с помощью функции

function GetProcAddress(module: hModule; procName: pChar): pointer;

В качестве первого параметра передается хэндл модуля загруженной DLL, в качестве второго – указатель на символьный идентификатор процедуры или числовой идентификатор, преобразованный к типу "указатель", например

pointer(10); - число 10, преобразованное к типу указатель

Возвращаемое значение – указатель на точку входа в процедуру или **nil**, если требуемая процедура отсутствует.

Использовать эту процедуру в программах на Паскале следует совместно с переменными процедурного типа (с указателями на процедуру), например так:

```
var Proc: procedure(a,b:integer); //процедура с двумя целыми параметрами
    hModule:THandle;
.....................
hModule:=LoadLibrary('testlib.dll');
@Proc:=GetProcAddress(hmodule,'ProcedureTwo'); //настройка
proc(1,2);  // вызов
FreeLibrary(hModule);
```

Имеется также возможность узнать хэндл уже загруженной библиотеки – например, если она загружена статическим образом при старте программы. Для этого служит функция

function GetModuleHandle(filename:PChar):HMODULE;

В качестве параметра необходимо передать имя файла уже загруженной библиотеки.

Использование ресурсов из DLL

Чтобы воспользоваться ресурсом из DLL, необходимо получить хэндл модуля этой DLL при помощи LoadLibrary или GetModuleHandle, после чего передать этот хэндл в обычную процедуру загрузки ресурса (LoadBitmap, LoadCursor и т.д.). Иногда DLL используются как чистые библиотеки ресурсов, т.е. не содержат исполнимых кодов вообще.

Чтобы загрузить библиотеку исключительно с целью доступа к ее ресурсам, рекомендуется применять функцию LoadLibraryEx примерно следующим образом:

```
hModule:=LoadLibraryEx('testlib.dll',0, LOAD_LIBRARY_AS_DATAFILE);
hBmp:=LoadBitmap(hModule,'bitmap1');
.................
DeleteObject(hBmp);
FreeLibrary(hModule);
```

При этом не выполняется код входа в DLL, что ускоряет загрузку.

Задание

Выполняется при самостоятельной подготовке:

1. Изучить теоретическую часть описания ЛР и материал соответствующих лекций.

2. По материалам Справки (Help) изучить описание следующих функций Windows и связанных с ними структур данных:

Функции:

Выполняется в лаборатории:

1. Включить компьютер. Запустить систему Delphi 2.0.

2. Загрузить исходный текст примера LAB7.PAS, изучить логику работы программы.

3. Откомпилировать и запустить пример. Изучить поведение созданных окон.

4. Запустить программу Resource Workshop, открыть файл ресурсов RES7.RES, сохранить его в виде файла .RC, изучить синтаксис текстового описания ресурсов.

5. Написать и отладить программу по индивидуальному заданию (см. ниже). Продемонстрировать результаты работы преподавателю.

6. Завершить работу с Delphi и Resource Workshop. Оставить компьютер включенным.

Варианты заданий:

В этой работе необходимо написать программу, пользующуюся ресурсами и функциями из DLL. Главное окно программы должно быть снабжено меню и собственным нестандартным значком. В одном из окон программы должен использоваться нестандартный курсор мыши. Все строки, битовые образы, курсоры, значки, используемые в программе, должны быть описаны в ресурсах.

Один из пунктов меню должен активизировать модальный диалог из ресурса, поведение которого соответствует заданию на ЛР №6.

Другой пункт меню должен активизировать "окно-заставку", фон которого заполнен битовым образом способом, указанным в задании ЛР №4, предусмотреть вывод номера бригады в этом окне.

Выход из программы также предусмотреть при помощи соответствующего пункта меню.

Еще один пункт меню должен приводить к вызову функции готовой DLL с именем LIB7.DLL в соответствии с вариантом (см. таблицу). В качестве параметра PChar необходимо передавать указатель на существующую строку, заканчивающуюся нулем.

№	Идентификатор	Параметры	Возвр. значение	Модель вызова	Способ загрузки DLL
1	1	a: integer; b: integer;	нет	C	статический
2	2	a: pchar; b: pchar;	integer	Pascal	динамический
3	'proc3'	a: pchar; b: char;	boolean	API	статический
4	'proc4'	a: pchar; b: pchar;	pointer	C	динамический
5	5	a: pchar; var b: integer;	нет	Pascal	статический
6	6	a: integer; b: integer;	integer	API	динамический
7	'proc7'	a: pchar; b: pchar;	char	C	статический
8	'proc8'	a: pchar; b: char;	smallint	Pascal	динамический
9	9	a: pchar; b: pchar;	boolean	API	статический
10	'proc10'	a: pchar; var b: integer;	нет	C	динамический

Лабораторная работа №8

Многозадачность. Многопоточность. Взаимодействие процессов

Работа выполняется в системе программирования Borland Delphi 2.0 в среде Windows 95. Использование средств визуального программирования и классов VCL не допускается, то есть Delphi используется просто как 32-разрядный компилятор с языка Паскаль для операционной системы Windows.

Теоретические положения

Windows 95 является 32-разрядной многозадачной операционной системой. Для 32-разрядных приложений Windows обеспечивает:

- плоскую (не сегментированную) модель виртуальной памяти;
- организацию для каждого процесса изолированного адресного пространства с возможностью адресации 4 Гб виртуальной памяти;
- возможность выполнения нескольких потоков в рамках каждого процесса;
- выполнение всех потоков всех 32-разрядных процессов в режиме вытесняющей многозадачности;
- поддержку средств синхронизации потоков – критических секций, семафоров, мутексов, событий;
- обмен данными между процессами.

Виртуальная память

Механизм виртуальной памяти позволяет решить проблему нехватки оперативной памяти в вычислительной системе. Он позволяет "расширить" объем оперативной памяти ВС за счет других видов памяти, имеющихся в системе, например – за счет дисковой памяти. При этом часть адресов (виртуальных адресов) соответствует адресам блоков информации, находящихся действительно в оперативной памяти, другие же адреса соответствуют информации, физически находящейся на диске.

Операционная система перехватывает обращение программы к виртуальным адресам и заботится о том, чтобы требуемый блок информации физически оказывался в оперативной памяти, при этом другие блоки перемещаются из памяти на диск. Кроме того, виртуальный адрес заменяется физическим адресом, соответствующим реальному положению адресуемого блока информации в оперативной памяти. Часть этих функций реализуется аппаратно процессором, часть – программно кодом операционной системы.

Плоская модель памяти Windows

ОС Windows 95 предназначена для выполнения на процессорах, совместимых с Intel 80386, в защищенном режиме. При этом поддерживаемый процессором механизм сегментации не используется, т.е. для обращения к любым данным или коду программа использует только 32-разрядное смещение. Сегментные регистры процессора при этом содержат нулевые значения и не могут быть изменены (попытка изменить сегментный регистр пользовательским процессом вызывает аппаратное прерывание и аварийное завершение программы операционной системой как выполнившей недопустимую команду).

32-разрядный адрес, используемый в любой машинной команде программы, не является физическим адресом, то есть процессор не выставляет его на шине адреса (разрядность которой – тоже 32 бита) для обращения к памяти. Этот адрес подвергается расшифровке в соот-

ветствии с аппаратно поддерживаемым процессором механизмом страничного преобразования.

Для каждого процесса операционной системой организуется собственное виртуальное адресное пространство. Это означает, в частности, что один и тот же адрес в контексте разных процессов как правило ссылается на различные физические адреса в памяти. За счет этого один пользовательский процесс физически не может испортить данные, с которыми работает другой процесс, так как их адресные пространства не пересекаются, и процесс просто не может задать такой адрес, чтобы обращение шло к данным другого процесса. С другой стороны, изолированность виртуальных адресных пространств процессов порождает проблемы, связанные с обменом данными между процессами – один процесс не может просто подготовить блок данных и передать в другой процесс указатель на этот блок, как это делалось в предыдущих версиях Windows (и возможно до сих пор, но только для 16-разрядных процессов). Windows API предоставляет специальные средства для обмена данными между процессами.

Адреса в виртуальном адресном пространстве пользовательских (не системных) процессов в Windows 95 распределяются следующим образом (речь идет не о выделении самой памяти, а о возможных значениях АДРЕСОВ):

$0..$3FFFFF – не используются 32-разрядными приложениями, могут использоваться для совместимости с программами DOS и Win16 (младшие 4Мбайта);

$400000..$7FFFFFFF – используются процессом для загрузки кодов и размещения данных (примерно 2 Гбайта);

$80000000..$BFFFFFFF – системная память, разделяемая всеми 32-разрядными приложениями (1 Гбайт);

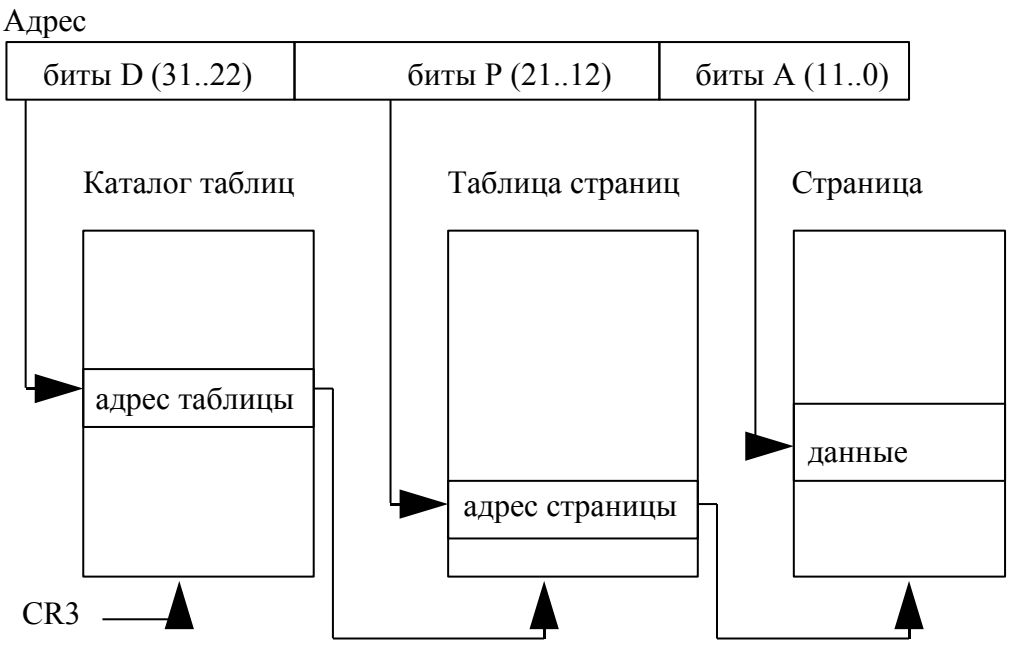
$C0000000..$FFFFFFFF – частная память процесса, используется ОС для обслуживания приложения (1 Гбайт).

Таким образом, приложения могут использовать для размещения данных и кода максимум два гигабайта адресного пространства из четырех возможных. Кроме того, приложение может иметь доступ к данным ОС через адреса в третьем гигабайте адресного пространства, что чревато нарушением работы ОС неверно работающим приложением. Модель памяти Windows NT не подразумевает такого разделения системной памяти приложениями, вследствие чего эта ОС значительно более устойчива к сбоям в приложениях.

Страничная организация виртуальной памяти

При страничной организации виртуальной памяти память выделяется процессам одинаковыми небольшими блоками. В процессорах Intel размер страниц составляет 4 кбайта. Каждая страница может находиться либо в оперативной памяти, либо на диске.

Для страничной организации виртуальной памяти операционная система поддерживает в памяти особые управляющие структуры, называемые таблицами страниц и каталогами страниц. Эти структуры аппаратно используются процессором для преобразования виртуальных адресов в физические. При этом биты 32-разрядного линейного адреса разделяются на три группы и используются следующим образом:



Каждая страница имеет размер 4 килобайта, и поэтому начинается с адреса, в котором младшие 12 бит нулевые (12 бит адресуют 4 кбайта).

ОС поддерживает таблицы страниц, в которых содержатся 32-разрядные дескрипторы страниц. Каждая таблица страниц сама по себе является страницей и занимает 4 кбайта. Старшие 20 бит дескриптора содержат старшие 20 бит физического адреса, с которого начинается страница (младшие 12 бит этого адреса нулевые). В оставшихся 12 битах дескриптора кодируются различные признаки, в частности – находится ли страница в памяти или на диске, можно ли производить запись в нее, была ли такая запись и т.д.

Кроме того, для каждого процесса ОС создает каталог таблиц страниц (тоже 4 кбайт, т.е. одна страница), с помощью которого процесс обращается к выделенным ему страницам памяти. Физический адрес этого каталога содержится в спецрегистре процессора CR3, и изменять его может только код, исполняемый в режиме ядра. Каждый элемент каталога таблиц содержит 32-разрядный дескриптор таблицы, в котором содержатся 20 старших бит ее адреса.

Когда в какой-либо машинной команде производится обращение по 32-разрядному адресу, процессор берет 20 разрядов адреса каталога из CR3, приписывает к ним десять битов поля D и два нуля и по полученному адресу извлекает дескриптор таблицы страниц. Затем берется 20 старших битов дескриптора, к ним приписываются 10 битов поля P, два нуля, и по полученному физическому адресу из таблицы страниц извлекается дескриптор страницы.

Если биты признаков дескриптора страницы содержат информацию о том, что страница находится в памяти, то происходит обращение к странице по физическому адресу, получаемому комбинацией 20 старших битов дескриптора и 12 битов поля A адреса. Если же страница отсутствует в памяти (находится на диске), то происходит аппаратное прерывание, обслуживая которое операционная система должна обеспечить подкачку нужной страницы с диска в память.

На самом деле процесс формирования физического адреса происходит без многократного обращения к таблицам дескрипторов в памяти, так как в процессоре есть средства для кэширования этой информации (хранения ее в сверхоперативной памяти внутри процессора).

Нетрудно видеть, что такой механизм управления памятью исключает проблему фрагментации физической памяти: страницы, образующие непрерывный блок в виртуальном адресном пространстве, могут быть как угодно разбросаны в физической памяти и загружаться при подкачке с диска в любое место. Вместе с тем, проблема фрагментации виртуальной памяти страничным механизмом не устраняется.

Сообщение WM_COPYDATA

Это сообщение позволяет передать блок информации от одного процесса другому. При этом в процессе обработки этого сообщения, и только в это время, передаваемый блок данных становится "видимым" из адресного пространства процесса-получателя этого сообщения и доступным для чтения. В документации к Windows сказано, что процесс-получатель не имеет права модифицировать информацию внутри блока, и он должен скопировать нужную информацию в самостоятельно выделенную область памяти, если эта информация нужна вне обработчика сообщения WM_COPYDATA.

Сообщение WM_COPYDATA всегда должно посылаться небуферизованным способом с помощью функции SendMessage. Отправлять его с помощью PostMessage недопустимо. Чтобы воспользоваться этим сообщением, нужно знать хэндл окна-получателя, относящегося к другому процессу, поэтому процессы предварительно должны "установить соединение" – например, с помощью широковещательной рассылки сообщений.

wParam - хэндл окна, отправившего данные; в принципе здесь может фигурировать хэндл любого окна процесса-отправителя.

lParam - указатель на структуру следующего вида:

```
TCopyDataStruct = packed record
  dwData: integer;
  cbData: dword;
  lpData: pointer;
end;
```

dwData – произвольное 32-разрядное значение, lpData – указатель на передаваемый блок данных, cbData – размер блока, адресуемого lpData.

Процесс-отправитель должен заполнить поля этой структуры, установив в lpData адрес блока данных, которые необходимо передать в другой процесс. Windows перед вызовом обработчика этого сообщения в другом процессе произведет необходимые действия по переносу данных в адресное пространство получателя, и в обработчике поле lpData окажется указывающим на блок передаваемых данных уже в адресном пространстве получателя. Этот указатель действителен только во время обработки сообщения, т.е. пока не завершилась функция SendMessage в процессе-отправителе.

С помощью этого сообщения можно передавать данные между любыми 16- и 32-разрядными приложениями.

Отображение файлов на память (File Mapping)

Само по себе отображение файлов на память работает следующим образом. Пользовательская программа просит ОС отобразить определенный дисковый файл в некоторый диапазон адресов своего виртуального адресного пространства, при этом создается особый управляющий объект, называемый отображением файла (file mapping). После этого при чтении и записи по адресам памяти, на которые отображен файл, программа читает и записывает содержимое соответствующего дискового файла. Это достигается за счет того, что страни-

цы памяти, соответствующие адресам отображения, выгружаются ОС не в системный файл подкачки, а в указанный файл; программа на самом деле читает и пишет в страницы памяти, в которые было считано содержимое файла при создании отображения и откуда информация будет возвращена в файл после закрытия отображения.

Сам по себе такой способ работы с файлом может иногда оказаться полезен, однако основное применение возможность отображения файлов в память находит не по прямому назначению (для работы с файлами), а для обмена информацией между приложениями. Это связано с тем, что один и тот же файл могут отобразить в память несколько приложений одновременно. При этом операционная система обеспечивает, чтобы изменения, сделанные в отображенном файле одним процессом, были отражены в образе файла, которую "видят" другие процессы. Фактически это достигается за счет того, что для отображения одного и того же файла в разных процессах используется одна и та же разделяемая физическая область памяти. При этом виртуальные адреса отображений в разных процессах могут не совпадать.

Для работы с отображением файла необходимы два объекта ОС: отображающий объект (file mapping) и образ файла (file view). Можно рассматривать образ файла как диапазон адресов виртуального адресного пространства процесса, используемых для работы с файлом, а отображающий объект – как страницы физической памяти, связанные с файлом. В связи с этим отображающий объект создается один раз в одном из взаимодействующих процессов, а образы файла – в каждом процессе с использованием существующего отображающего объекта.

Для создания отображающего объекта нужен хэндл открытого файла. Обычно файл специально создается для этой цели с помощью функции CreateFile, при этом желательно, чтобы создающий процесс создавал файл с эксклюзивным доступом (другие процессы не могли бы его открыть/испортить). У этой функции достаточно много параметров (см. Справку по Win32 API), возвращает она хэндл файла:

```
hFile := CreateFile(
      'myfile.map', // имя файла
      GENERIC_READ or GENERIC_WRITE, // доступ на чтение-запись
      0, // эксклюзивный доступ
      0, // атрибуты защиты, не поддерживается Windows 95, только NT
      CREATE_ALLWAYS, // создавать заново
      FILE_ATTRIBUTE_NORMAL);
```

Затем может быть создан отображающий объект:

```
hMapping := CreateFileMapping(
      hFile, // хэндл открытого файла
      0, // атрибуты защиты, не поддерживается Windows 95, только NT
      PAGE_READ_WRITE,
      0, 4096, // 4096 байт, если 0 - размер равен размеру файла
      'MyMapping' // имя-идентификатор объекта
      );
```

После этого может быть создан образ файла в адресном пространстве процесса, функция возвращает указатель на начало блока памяти размером, в нашем случае, 4096 байт:

```
pData := MapViewOfFile(
      hMapping, // хэндл отображающего объекта
      FILE_MAP_ALL_ACCESS, // доступ на чтение-запись
      0,0,0 // доступ ко всему отображению (можно задать часть)
      );
```

Далее можно производить в области, адресуемой указателем pData действия по чтению и записи информации. Например, можно переписать оттуда 256 байт в короткую строку с именем S:

Move(pData^, S, 256);

Другие процессы должны не создавать отображающий объект, а получать хэндл готового по имени:

```
hMapping := OpenFileMapping(0,false,'MyMapping');
```

Далее тоже вызывается MapViewOfFile.

При использовании данного механизма для обмена данными имеется возможность отказаться от использования собственного файла-основы и отображать системный файл подкачки. Для этого в функции CreateFileMapping в качестве первого параметра можно задать –1 (т.е. $FFFFFFFF). В этом случае обязательно необходимо задавать размер отображения.

Образ файла должен уничтожаться после использования функцией

```
UnmapViewOfFile(pData);
```

в качестве аргумента указывается указатель на образ файла, полученный с помощью MapViewOfFile.

Отображающий объект и сам файл (если он создавался) закрываются с помощью функции CloseHandle:

```
CloseHandle(hMapping);
CloseHandle(hFile);
```

Обычно разделение памяти осуществляется между тесно связанными программами, входящими в состав одного пакета. Например, некоторые сервисные функции пакета могут быть реализованы в отдельных программах-утилитах, работающих как самостоятельно, так и под управлением главной программы пакета. В этом случае проблемы "кому создавать отображающий объект" обычно не возникает – его создает главная программа, а вызываемая ей программа-утилита использует готовый объект через OpenFileMapping.

Если же разделение памяти должно осуществляться между равноправными приложениями (например – разными экземплярами одного и того же приложения), то можно использовать следующий подход:

1. Приложение пытается открыть объект с помощью OpenFileMapping.

2. Если возвращено нулевое значение, то объект не существует, поэтому его надо создать именно в этом экземпляре приложения, а затем удалить при помощи CloseHandle.

Обмен сообщениями между приложениями

Windows предоставляет возможность посылать сообщения между приложениями. При этом программист имеет возможность реализовать любой удобный для его программных продуктов протокол связи между процессами. С помощью посылки сообщений реализован также стандартный протокол обмена данными DDE (см. описание в отдельном файле). Обычно с помощью сообщений передают небольшие объемы данных, для кодирования которых достаточно двух 4-байтовых параметров сообщения.

При реализации собственного протокола обмена сообщениями необходимо учитывать, что Windows "согласится" доставлять только сообщения, коды которых лежат в диапазоне C000h .. FFFFh.

Хотя программист может "волевым решением" установить, что его программы будут пользоваться сообщениями, допустим, с кодом D000h и E000h, такой подход нельзя назвать удачным: никто не гарантирует, что какие-то программы параллельно не посылают друг другу сообщения с такими кодами. Поэтому Windows в виде функции RegisterWindowMessage

предоставляет возможность регистрировать сообщения по уникальному строковому идентификатору. Вероятность того, что различные группы приложений используют одни и те же строковые идентификаторы сообщений можно сделать достаточно низкой, если использовать в идентификаторах, например, название фирмы-производителя программного продукта.

Например, приложение может выполнить команду:

idMSG1:= RegisterWindowMessage('Borland_Data_exchange_message');

При этом функция возвращает значение, однозначно соответствующее указанной строке, т.е. если другое приложение вызовет эту же функцию с такой же строкой в качестве параметра, то будет возвращено то же число, а для другой строки число гарантированно будет другим. Если по каким-то причинам зарегистрировать сообщение не удается (например, слишком длинная строка, хотя в документации такие ограничения не упомянуты), то возвращается 0, иначе – число в диапазоне C000h .. FFFFh. Зарегистрированные сообщения остаются зарегистрированными до завершения сеанса работы в Windows (выключения или перезагрузки компьютера).

Часто для начального установления связи одно из приложений широковещательно рассылает сообщение-приветствие, в котором указывается хэндл окна-отправителя. Широковещательная рассылка достигается при указании в первом параметре SendMessage константы HWND_BROADCAST вместо хэндла окна. При получении приветствия (такое сообщение должно быть зарегистрировано как инициатором связи, так и предполагаемым партнером) другое приложение отвечает сообщением-подтверждением, в котором также указывается хэндл окна, которому следует направлять дальнейшие сообщения в процессе связи.

Например, два приложения зарегистрировали одни и те же сообщения:

```
msg1:= RegisterWindowMessage('Запрос');
msg2:= RegisterWindowMessage('Ответ');
msg3:= RegisterWindowMessage('Данные');
```

Затем первое приложение рассылает сообщение-запрос:

```
SendMessage(HWND_BROADCAST, msg1, hClientWindow,0);
```

При получении такого сообщения (его получат все окна первого плана, т.е. не имеющие окон-собственников, всех прилохений) второе приложение запоминает хэндл запрашивавшего окна (в нашем случае он передавался в wParam) и отвечает, указывая хэндл своего окна, например так:

```
hOtherWindow:=wparam;
PostMessage(wparam, msg2, hServerWindow, 0);
```

Приняв такое сообщение, окно hClientWindow первого приложения готово обмениваться сообщениями с окном hServerWindow второго приложения, причем в каждом приложении известен хэндл окна-партнера из другого приложения.

```
hOtherWindow:=wparam;
```

И теперь в любом приложении может быть послано сообщение:

```
PostMessage(hOtherWindow,msg3, 10,20);
```

где hOtherWindow – хэндл окна-партнера, сохраненный при установлении связи.

При двустороннем обмене сообщениями всегда предпочтительно пользоваться функцией PostMessage (если нет особых причин использовать SendMessage), так как при этом исключается рекурсивный вызов оконной процедуры. Дело в том, что если в ответ на SendMessage одного приложения второе приложение пошлет ответ через SendMessage, то оконная процедура в первом приложении запустится еще раз (рекурсивно), хотя предыдущий код,

посылавший SendMessage, еще не завершен. Это вполне допустимо, однако при плохом понимании программистом особенностей рекурсии способствует появлению логических ошибок в программе.

Буфер обмена

Приложения в принципе могут обмениваться информацией, помещая ее в системный буфер обмена и считывая оттуда. Однако это некорректно по отношению к пользователю. Дело в том, что буфер обмена предназначен для того, чтобы пользователь компьютера по своей инициативе помещал туда те или иные данные или считывал их оттуда (точнее – давал соответствующие инструкции используемым приложениям). Если приложения начнут записывать данные по своему усмотрению, а не по желанию пользователя, то это будет мешать его (пользователя) удобной и нормальной работе.

Для работы с буфером обмена существуют функции (подробности Clipboard overview в Справке):

OpenClipboard, EmptyClipboard, SetClipboardData, GetClipboardData, CloseClipboard, GetClipboardOwner, RegisterClipboardFormat, IsClipboardFormatAvailable и др.

Организация нескольких потоков в рамках процесса

Windows позволяет исполнять в рамках одного процесса несколько участков кода (потоков, нитей – threads) в режиме вытесняющей многозадачности. При этом при создании процесса порождается единственный (главный) поток, в котором могут быть выполнены действия для порождения дополнительных потоков. Рекомендуется строить программу так, чтобы все операции диалога с пользователем были сосредоточены в одном (главном) потоке программы, т.е. только там рекомендуется порождать все окна. В дополнительные потоки рекомендуется выносить длительные операции, не требующие общения с пользователем, например – сложный итеративный расчет или долгую операцию, связанную с обменом большим объемом информации с медленным устройством вроде дисковода или модема.

Для каждого потока Windows создает очередь сообщений, как только в потоке будет произведен вызов функции модулей USER или GDI. Это означает, что если поток не создает окон, нигде не рисует и не вызывает других функций USER и GDI, то очереди сообщений у него нет.

Все окна, созданные во время работы потока, получают сообщения именно через его очередь сообщений. Это означает, что в каждом потоке, создающем окна, должен быть СВОЙ цикл обработки сообщений, так как через главный цикл обработки сообщений программы идут только сообщения, относящиеся к окнам, созданным в главном потоке.

Если поток согласно логике работы программы ждет наступления какого-либо события, его следует переводить в состояние ожидания. Например, когда поток ждет прихода очередного сообщения, он переводится в состояние ожидания функцией GetMessage. Имеется возможность описывать в программе собственные события, управлять их состоянием (событие наступило или не наступило) и переводить потоки в состояние ожидания до тех пор, пока, например, другой поток не установит некоторое событие в состояние "наступило".

Поток создается при вызове функции

```
function CreateThread(
        lpThreadAttributes: Pointer;
        dwStackSize: DWORD;
```

lpStartAddress: Pointer;
lpParameter: Pointer;
dwCreationFlags: DWORD;
var lpThreadId: DWORD): THandle;

Назначение параметров:

lpThreadAttributes – атрибуты защиты, в Windows 95 не используется, передается nil.

dwStackSize – размер стека; если 0, то выделяется такой же стек, как у главного потока; при необходимости размер стека в дальнейшем автоматически увеличивается.

lpStartAddress – указатель на процедуру потока.

lpParameter – указатель (в принципе – любое 32-разрядное значение), который передается процедуре потока при запуске.

dwCreationFlags – равен 0 или CREATE_SUSPENDED, в последнем случае поток порождается в состоянии ожидания, из которого выводится функцией ResumeThread.

lpThreadId – указатель на 32-разрядную переменную, в которую будет записан идентификатор (не хэндл!) потока, нужный некоторым функциям управления потоками.

Возвращаемое значение – хэндл потока (или 0 при неудаче).

Функция потока объявляется как:

function MyThreadProc(data: pointer): dword; **stdcall**;

Поток завершается:

- когда функция потока завершается, при этом возвращаемое ею значение считается кодом возврата;

- когда внутри функции потока (или вызываемых ею функций) выполняется вызов функции ExitThread(code), где code – 32-разрядный код возврата;

- когда в любом потоке вызывается функция TerminateThread с указанием хэндля завершаемого потока и кода возврата; это является аварийным завершением, при котором не освобождаются загруженные завершаемым потоком DLL, занятый стек и т.д.

После завершения потока код возврата можно узнать с помощью функции

GetExitCodeThread(hThread, status);

где status – 32-разрядная переменная, в которую будет передан код возврата. Если поток еще исполняется (не завершен), то будет возвращено значение STILL_ACTIVE.

После завершения потока соответствующий системный объект продолжает существовать, и его хэндл можно удалить при помощи функции

CloseHandle(hThread);

Хэндл потока действителен только в рамках процесса, в котором он существует. При завершении процесса все потоки завершаются, их хэндлы освобождаются. В отличие от этого идентификатор потока действует глобально во всей системе.

Функция ExitProcess(exitCode) приводит к завершению всех потоков процесса и самого процесса.

Средства синхронизации потоков

Критические секции

Для защиты критических участков кода, выполняющихся в различных потоках ОДНОГО ПРОЦЕССА Windows поддерживает механизм критических секций. Так как критическая секция организуется с помощью особой структуры, располагаемой в изолированном адресном пространстве процесса, этот механизм непригоден для синхронизации параллельно выполняющихся процессов.

Чтобы воспользоваться критической секцией, программа должна создать экземпляр переменной типа TRTLCriticalSection (захватив под него память из кучи или просто создав глобальную переменную этого типа):

var section: TRTLCriticalSection;

Информация внутри структуры должна быть предварительно проинициализирована с помощью вызова функции

InitializeCriticalSection(section);

Далее потоки могут пытаться занять критическую секцию с помощью вызовов

EnterCriticalSection(section)

или

if TryEnterCriticalSection(section) then ...

Первая функция просто реализует захват критической секции вызывающим ее потоком, при занятости КС поток переводится в состояние ожидания и ждет освобождения КС.

Вторая функция вместо перехода в состояние ожидания возвращает логическое значение FALSE, если критическая секция занята (и входит в КС с возвратом TRUE, если КС свободна).

После завершения действий, выполняемых в КС, поток должен освободить КС, вызвав

LeaveCriticalSection(section);

Критических секций в программе может быть сколько угодно.

Если КС используется в программе только на одном из этапов ее работы, то занимаемые КС системные ресурсы можно освободить, вызвав функцию

DeleteCriticalSection(section);

Защищенные целые операции

Windows обеспечивает функции для производства простых операций с целыми переменными, в ходе выполнения которых гарантируется, что доступ к изменяемым переменным из других потоков производиться не будет. Если переменные находятся в разделяемой памяти, то их можно использовать для синхронизации процессов.

См. Справку по следующим функциям:

InterlockedIncrement, InterlockedDecrement, InterlockedExchange, InterlockedCompareExchange, InterlockedExchangeAdd.

Функции ожидания

Windows API содержит ряд функций, с помощью которых поток может быть переведен в состояние ожидания на определенное время или до наступления некоторых системных событий. Далее рассмотрено лишь ограниченное количество таких функций.

procedure Sleep(milliseconds:integer);

- Эта функция переводит вызывающий ее поток в состояние ожидания на указанное количество миллисекунд (32-разрядное целое). При указании в качестве параметра 0 происходит просто досрочная передача управления другим потокам, находящимся в состоянии готовности. Поток, переведенный в состояние ожидания с помощью этой функции, не может быть возобновлен до истечения указанного времени. Это означает, что если в этом потоке создавались окна и какому-либо окну посылаются сообщения, пусть даже с помощью SendMessage, эти сообщения не будут обработаны до завершения указанного времени.

function WaitForSingleObject(hObject:THandle; TimeOut:integer):dword;

- Функция ожидает перехода объекта синхронизации с хэндлом hObject (семафора, мутекса, события) во взведенное состояние. С помощью этой функции также можно ожидать завершения потока или процесса, передав соответствующий хэндл в качестве объекта ожидания. Выход из состояния ожидания производится также по истечении времени в миллисекундах, указываемого во втором параметре. Можно задать бесконечный тайм-аут при помощи константы INFINITE. При указании 0 в качестве тайм-аута сначала проверяется состояние объекта, и если он взведен, то это указывается в коде возврата. Функция возвращает значение, позволяющее судить о причинах выхода из состояния ожидания:

WAIT_OBJECT_0 – объект перешел во взведенное состояние (процесс или поток переходят в такое состояние после завершения);

WAIT_TIMEOUT – истек тайм-аут;

WAIT_ABANDONED – специфично для мутексов (см. ниже): поток, владевший мутексом, не освободил его в явном виде, но завершился, вследствие чего мутекс освободился и передан данному потоку.

function WaitForMultipleObjects(nCount: DWORD; lpHandles: pointer; bWaitAll: BOOL; dwMilliseconds: DWORD): DWORD;

Во втором параметре передается указатель на массив хэндлов объектов, в первом – количество хэндлов в этом массиве (не должно превышать MAXIMUM_WAIT_OBJECTS), третий параметр определяет, должны ли для окончания ожидания быть взведены все объекты массива (TRUE), или достаточно взведения лишь одного из них (FALSE). В четвертом параметре указывается тайм-аут ожидания. Возвращаемое значение:

WAIT_TIMEOUT – истек тайм-аут;

WAIT_OBJECT_0 .. WAIT_OBJECT_0 + (nCount–1) – объект с соответствующим номером перешел во взведенное состояние (процесс или поток переходят в такое состояние после завершения);

WAIT_ABANDONED_0 .. WAIT_ABANDONED_0 + (nCount–1) – специфично для мутексов, имело место для мутекса, хэндл которого указан в массиве в соответствующей позиции.

Семафоры

Windows поддерживает семафоры как объекты операционной системы. Семафоры могут использоваться для синхронизации между потоками. Семафор в Windows имеет целое значение от 0 до максимального, объявляемого при создании семафора. Семафор считается взведенным (не требует ожидания), когда его значение отлично от нуля. Если значение семафора равно нулю, то обращение к функции ожидания с указанием его хэндла приводит к переводу потока в состояние ожидания. При любом обращении к функции ожидания с указанием хэндла взведенного семафора его (семафора) значение автоматически уменьшается.

Пример создания семафора:

```
hSema:=CreateSemaphore(
          nil, // атрибуты защиты
          1, // начальное значение
          1, // максимальное значение
          'MySemaphore' // имя семафора
          );
```

Уменьшение значений семафора с возможным переходом в состояние ожидания осуществляется функциями ожидания, например

WaitForSingleObject(hSema,INFINITE);

Увеличение значения семафора производится функцией

ReleaseSemaphore(hSema,1,nil);

- увеличение на единицу, в последнем параметре можно передать указатель на 32-разрядную переменную, в которую будет помещено предыдущее значение семафора.

Получить хэндл уже созданного семафора по имени (например, в зависимом процессе) можно при помощи функции

hSema := OpenSemaphore(SEMAPHORE_ALL_ACCESS, true, 'MySemaphore');

Также можно вызвать CreateSemaphore, которая в случае существования семафора с тем же именем вернет хэндл уже существующего объекта, при этом начальное и максимальное значение будут проигнорированы.

Удаление семафора:

CloseHandle(hSema);

Мутексы

Мутексы (от mutual exclusion – взаимное исключение) представляют собой объекты, владеть которыми может только один поток. При задании хэндла мутекса в функции ожидания, поток либо переходит в состояние ожидания, если мутекс захвачен другим потоком, либо захватывает мутекс и продолжает свое выполнение. Действия с мутексами во многом аналогичны действиям с семафорами.

См. функции:

CreateMutex, OpenMutex, ReleaseMutex, CloseHandle.

События

События (events) представляют собой объекты, которые могут взводиться и сбрасываться произвольным образом по усмотрению программиста. При этом имеется возможность создавать события, которые автоматически сбрасываются при завершении функции ожидания, а также события, которые необходимо сбрасывать в явном виде. Действия с событиями аналогичны действиям с семафорами и мутексами, см. функции

CreateEvent, OpenEvent, SetEvent, ResetEvent, CloseHandle.

Задание

Выполняется при самостоятельной подготовке:

1. Изучить теоретическую часть описания ЛР и материал соответствующих лекций.

2. По материалам Справки (Help) изучить описание функций и структур данных, на которые имеются ссылки в описании работы.

Выполняется в лаборатории:

1. Включить компьютер. Запустить систему Delphi 2.0.

2. Загрузить исходный текст примера

3. Откомпилировать и запустить пример. Изучить поведение созданных окон.

4. Написать и отладить программу в соответствии с индивидуальным заданием. Продемонстрировать результаты работы преподавателю.

5. Завершить работу с Delphi. Оставить компьютер включенным.

Варианты заданий:

1. "Телетайп". Два различных приложения. В одном имеется окно редактирования (edit), в котором пользователь набирает текст. Набираемые символы должны параллельно отображаться в другом окне. Для передачи символов и установления связи использовать межпроцессные сообщения (любые). При отсутствии запущенного приложения-сервера выдавать сообщение об ошибке.

2. Многопоточная программа, аналогичная лаб.2: главный поток обслуживает интерфейс пользователя, два дочерних потока рисуют в клиентской области окна случайные линии и окружности. Для запуска и остановки рисования предусмотреть кнопки.

3. "Массовая рассылка". Приложение построено следующим образом: имеется окно класса edit для ввода текста и кнопка "отправить". Может быть запущено произвольное количество экземпляров приложения. При нажатии кнопки "отправить" набранный текст должен быть скопирован в окна ввода всех запущенных экземпляров этого приложения. Размер текста ограничен 2000 символов. Использовать отображение файлов на память и широковещательную рассылку сообщения "обновить".

4. Имеется массив из 10 строк, строки состоят из 20 символов, символы в каждой строке одинаковые. Главный поток раз в секунду отображает текущее состояние массива и обеспечивает интерфейс пользователя (создание и закрытие окна). Поток B постоянно циклически сдвигает строки в массиве: 2-я становится 1-й, 3-я – 2-й, 1-я – 20-й. Поток C постоянно случайным образом меняет местами две строки в массиве. С помощью применения семафоров добиться корректной работы с массивами. Все потоки должны нормально завершаться при закрытии окна.

5. Имеется массив из 10 строк, строки состоят из 20 символов, символы в каждой строке одинаковые. Главный поток раз в секунду отображает текущее состояние массива и

обеспечивает интерфейс пользователя (создание и закрытие окна). Поток B постоянно циклически сдвигает строки в массиве: 2-я становится 1-й, 3-я – 2-й, 1-я – 20-й. Поток C постоянно случайным образом меняет местами две строки в массиве. С помощью применения мутексов добиться корректной работы с массивами. Все потоки должны нормально завершаться при закрытии окна.

6. Имеется массив из 10 строк, строки состоят из 20 символов, символы в каждой строке одинаковые. Главный поток раз в секунду отображает текущее состояние массива и обеспечивает интерфейс пользователя (создание и закрытие окна). Поток B постоянно циклически сдвигает строки в массиве: 2-я становится 1-й, 3-я – 2-й, 1-я – 20-й. Поток C постоянно случайным образом меняет местами две строки в массиве. С помощью применения механизма событий. Все потоки должны нормально завершаться при закрытии окна.

7. Имеется массив из 10 строк, строки состоят из 20 символов, символы в каждой строке одинаковые. Главный поток раз в секунду отображает текущее состояние массива и обеспечивает интерфейс пользователя (создание и закрытие окна). Поток B постоянно циклически сдвигает строки в массиве: 2-я становится 1-й, 3-я – 2-й, 1-я – 20-й. Поток C постоянно случайным образом меняет местами две строки в массиве. С помощью применения механизма критических секций. Все потоки должны нормально завершаться при закрытии окна.

8. "Надзиратель". Взаимодействие программы-сервера и программы-клиента строится следующим образом: при запуске программы-клиента она посылает широковещательное сообщение с целью установить связь с сервером. Если сервер есть, он в ответ сообщает клиенту его номер, который клиент должен отобразить в рабочей области своего окна. Сервер отображает текущее количество клиентов. При завершении клиент посылает серверу уведомление. Считается, что сервер не может быть завершен раньше клиента (такую ситуацию обрабатывать не требуется).

9. "Надзиратель". Взаимодействие программы-сервера и программы-клиента строится следующим образом: при запуске программы-клиента она посылает широковещательное сообщение с целью установить связь с сервером. Если сервер есть, он в ответ сообщает клиенту его имя (строку длиной до 255 символов), которое клиент должен отобразить в заголовке своего окна. Сервер отображает текущее количество клиентов. При завершении клиент посылает серверу уведомление. Считается, что сервер не может быть завершен раньше клиента (такую ситуацию обрабатывать не требуется). Использовать для передачи строк отображение файла на память.

10. "Надзиратель". Взаимодействие программы-сервера и программы-клиента строится следующим образом: при запуске программы-клиента она посылает широковещательное сообщение с целью установить связь с сервером. Если сервер есть, он в ответ сообщает клиенту его имя (строку длиной до 255 символов), которое клиент должен отобразить в заголовке своего окна. Сервер отображает текущее количество клиентов. При завершении клиент посылает серверу уведомление. Считается, что сервер не может быть завершен раньше клиента (такую ситуацию обрабатывать не требуется). Использовать для передачи строк сообщение WM_COPYDATA.