WindowsNT и её API

Введение

Microsoft (r) Windows (c) NT (tm) - линейка операционных систем производства корпорации Microsoft, полученная путем противоестественного скрещивания ядра NT с юзерспейсом Windows. Изначально NT планировалась как операционная система для вычислительных систем на базе новой разработки Intel - процессора N10 (N-Ten, откуда и название), но ввиду отмены проекта ядро NT было перенесено на x86 и дополнено пользовательской средой Windows.

Архитектура Windows NT

Архитектура Windows NT имеет модульную структуру и состоит из двух основных уровней - компоненты, работающие в режиме пользователя, и компоненты режима ядра. Программы и подсистемы, работающие в режиме пользователя, имеют ограничения на доступ к системным ресурсам. Режим ядра имеет неограниченный доступ к системной памяти и внешним устройствам. Ядро системы NT называют гибридным ядром или макроядром. Архитектура включает в себя само ядро, уровень аппаратных абстракций (HAL), драйверы и ряд служб (Executives), которые работают в режиме ядра (Kernel-mode drivers) или в пользовательском режиме (User-mode drivers).

Пользовательский режим Windows NT состоит из подсистем, передающих запросы ввода-вывода соответствующему драйверу режима ядра посредством менеджера ввода-вывода. Есть две подсистемы на уровне пользователя: подсистема окружения (запускает приложения, написанные для разных операционных систем) и интегрированная подсистема (управляет особыми системными функциями от имени подсистемы окружения). Режим ядра имеет полный доступ к аппаратной части и системным ресурсам компьютера.

Программные интерфейсы

Native API

Для прикладных программ системой Windows NT предоставляется несколько наборов API. Основной из них — так называемый «родной» API (NT Native API), реализованный в динамически подключаемой библиотеке ntdll.dll и состоящий из двух частей: системные вызовы ядра NT (функции с префиксами Nt и Zw, передающие выполнение функциям ядра ntoskrnl.exe с теми же названиями) и функции, реализованные в пользовательском режиме (с префиксом Rtl). Часть функций второй группы используют внутри себя системные вызовы; остальные целиком состоят из непривилегированного кода, и могут вызываться не только из кода пользовательского режима, но и из драйверов. Кроме функций Native API, в ntdll также включены функции стандартной библиотеки языка Си.

Программы, выполняющиеся до загрузки подсистем, обеспечивающих работу остальных API ОС Windows NT, ограничены использованием Native API. Например,

программа autochk, проверяющая диски при загрузке ОС после некорректного завершения работы, использует только Native API.

DOS и Win16

Чтобы обеспечить двоичную совместимость с существующими программами для предыдущих семейств ОС от Microsoft, в Windows NT была добавлена программа-эмулятор ntvdm, реализующая VDM (виртуальную DOS-машину), внутри которой может выполняться программа для DOS. Для каждой выполняемой DOS-программы создаётся собственная VDM, тогда как несколько 16-битных Windows-программ могут выполняться в отдельных потоках внутри одной VDM, которая в этом случае играет роль подсистемы. Для того, чтобы внутри VDM можно было выполнять программы для Windows, в неё сначала должна быть загружена программа wowexec, устанавливающая связь VDM с платформой WOW («Windows on Win32»), позволяющей использовать 16-битные приложения для Windows наравне с 32-битными. Сама программа-эмулятор ntvdm выполняется внутри подсистемы Win32, что позволяет Win32-программам обращаться к окнам DOS-программ как к обычным консольным окнам, а к окнам Win16-программ - как к обычным графическим окнам.

В связи с аппаратными ограничениями 64-битных платформ, поддержка VDM и WOW была исключена из 64-битных версий Windows, запуск 16-битных программ средствами системы на них невозможен, но возможно использование эмуляторов, таких как DOSBox. Основным API этих версий Windows NT является 64-битная версия Win32 API; для запуска 32-битных программ используется технология WOW64, аналогичная традиционной WOW.

WSL

Windows Subsystem for Linux (WSL) - слой совместимости для запуска Linux-приложений (двоичных исполняемых файлов в формате ELF) в ОС Windows 10. В рамках сотрудничества компаний Майкрософт и Canonical стало возможным использовать оригинальный образ ОС Ubuntu 14.04 для непосредственного запуска поверх WSL множества инструментов и утилит из этой ОС без какой-либо виртуализации. WSL предоставляет интерфейсы, во многом совместимые с интерфейсами ядра Linux; однако подсистема WSL была полностью разработана корпорацией Майкрософт и не содержит в себе каких-либо исходных кодов ядра Linux. WSL запускает многие немодифицированные приложения, работающие в пространстве пользователя, в частности, оболочку bash, утилиты sed, awk, интерпретаторы языков программирования Ruby, Python, и т.д.

По состоянию на 2016 год, подсистема не поддерживает запуск графических приложений Линукс (использующих графические пользовательские интерфейсы GUI на базе X11) и приложений, требующих нереализованных интерфейсов ядра Linux. Ряд пользователей проводит эксперименты по запуску графических приложений с внешним сервером оконной системы X11, например VcXsrv или Xming. WSL использует меньше ресурсов, чем полная виртуализация, и стала наиболее простым путем запуска многих Linux-приложений на ОС Windows. Приложения Windows и Linux, запущенные через WSL, имеют доступ ко всем файлам пользователя.

POSIX и OS/2

В отличие от большинства свободных Unix-подобных ОС, Windows NT сертифицирована институтом NIST на совместимость со стандартом POSIX.1, и даже с более строгим стандартом FIPS 151-2. Библиотекой psxdll экспортируются стандартные функции POSIX, а также некоторые функции Native API, не имеющие аналогов в POSIX - например, для работы с кучей, со структурными исключениями, с Unicode. Внутри этих функций используются как Native API, так и LPC-вызовы в подсистему psxss, являющуюся обычным Win32-процессом.

Обе эти подсистемы, необязательные для работы большинства приложений, были удалены в Windows XP и последующих выпусках Windows. При помощи манипуляций с реестром их можно было отключить и в предыдущих версиях Windows NT, что рекомендовалось специалистами по компьютерной безопасности в целях сокращения поверхности атаки компьютерной системы.

Win32 API

Чаще всего прикладными программами для Windows NT используется Win32 API - интерфейс, созданный на основе API ОС Windows 3.1, и позволяющий перекомпилировать существующие программы для 16-битных версий Windows с минимальными изменениями исходного кода. Совместимость Win32 API и 16-битного Windows API настолько велика, что 32-битные и 16-битные приложения могут свободно обмениваться сообщениями, работать с окнами друг друга и т.д. Кроме поддержки функций существовавшего Windows API, в Win32 API был также добавлен ряд новых возможностей, в том числе поддержка консольных программ, многопоточности, и объектов синхронизации, таких как мьютексы и семафоры. Документация на Win32 API входит в состав Microsoft Platform SDK и доступна онлайн.

Библиотеки поддержки Win32 API в основном названы так же, как системные библиотеки Windows 3.x, с добавлением суффикса 32: это библиотеки kernel32, advapi32, gdi32, user32, comctl32, comdlg32, shell32 и ряд других. Функции Win32 API могут:

- либо самостоятельно реализовывать требуемую функциональность в пользовательском режиме,
- либо вызывать описанные выше функции Native API,
- либо обращаться к подсистеме csrss посредством механизма LPC,
- либо осуществлять системный вызов в библиотеку win32k, реализующую необходимую для Win32 API поддержку в режиме ядра.

Четыре перечисленных варианта могут также комбинироваться в любом сочетании: например, функция Win32 API WriteFile обращается к функции Native API NtWriteFile для записи в дисковый файл, и вызывает соответствующую функцию csrss для вывода в консоль.

Поддержка Win32 API включена в семейство ОС Windows 9x; кроме того, она может быть добавлена в Windows 3.1x установкой пакета Win32s. Для облегчения переноса существующих Windows-приложений, использующих для представления строк MBCS-кодировки, все функции Win32 API, принимающие параметрами строки, были созданы

в двух версиях: функции с суффиксом A (ANSI) принимают MBCS-строки, а функции с суффиксом W (wide) принимают строки в кодировке UTF-16. В Win32s и Windows 9x поддерживаются только A-функции, тогда как в Windows NT, где все строки внутри ОС хранятся исключительно в UTF-16 (если точнее, то в ее упрощенной вариации - UCS-2), каждая A-функция просто преобразует свои строковые параметры в Юникод и вызывает W-версию той же функции. В поставляемых H-файлах библиотеки также определены имена функций без суффикса, и использование A- либо W-версии функций определяется опциями компиляции, а в модулях Delphi до 2010 версии, например, они жёстко завязаны на варианты с суффиксом A. При этом важно отметить, что большинство новых функций, появившихся в Windows 2000 или более поздних ОС семейства Windows NT, существуют только в Unicode-версии, потому что задача обеспечения совместимости со старыми программами и с ОС Windows 9x уже не стоит так остро, как раньше.

Кроме изначальной и эталонной реализации WinAPI от Microsoft, существуют также и сторонние его реализации:

- Проект Wine https://winehq.org направлен на создание прослойки, позволяющей путем трансляции системных вызовов WinAPI в вызовы POSIX запускать приложения, изначально предназначенные для Windows, в среде UNIX-подобных операционных систем.
- Проект ReactOS https://reactos.org ставит своей целью создание операционной системы, полностью (на уровне API и ABI) совместимой с Microsoft Windows NT. Многие ее разработчики являются известными специалистами по внутреннему устройству ОС Windows, некоторые впоследствии переходили на работу в Microsoft. По возможности, в данном курсе будут приводиться примеры или ссылки на исходный код этой ОС.

Windows API

Функции, предоставляемые Windows API можно разбить на восемь категорий.

Основные сервисы (Base Services)

Предоставляет доступ к базовым ресурсам, доступным в Windows. Сюда относятся такие вещи, как файловые системы, устройства, процессы, потоки и обработка ошибок. Эти функции расположены в kernel.exe, krnl286.exe или krnl386.exe в 16-разрядных версиях Windows, и в kernel32.dll в 32- и 64-разрядных.

Дополнительные сервисы (Advanced Services)

Предоставляют доступ к функциям за пределами ядра. Сюда относятся реестр Windows, выключение/перезапуск системы (или их отмена), запуск/остановка/создание службы Windows, управление учетными записями пользователей. Эти функции располагаются в advapi32.dll в 32- и 64-разрядных Windows.

Интерфейс графических устройств (Graphics Device Interface)

Предоставляет функции для вывода графического содержимого на мониторы, принтеры и другие устройства вывода. Расположены в gdi.exe в 16-разрядных Windows.

Пользовательская часть в 32- и 64-разрядных Windows расположена в библиотеке gdi32.dll, поддержка GDI на уровне ядра обеспечивается драйвером win32k.sys, который взаимодействует напрямую с графическим драйвером.

Пользовательский интерфейс (User Interface)

Предоставляет функции для создания и управления экранными окнами и большинством базовых элементов, таких, как кнопки и полосы прокрутки, получения ввода с мыши и клавиатуры, а также другие функции, связанные с подсистемой графического пользовательского интерфейса (GUI) Windows. Данная функциональная единица размещена в user.exe на 16-разрядных Windows и user32.dll на 32- и 64-разрядных. Начиная с Windows XP, основные элементы управления вынесены в comctl32.dll, вместе со всеми остальными.

Библиотека общих диалоговых окон (Common Dialog Box Library)

Предоставляет приложениям стандартные диалоговые окна открытия и сохранения файлов, выбора цвета и шрифта и т.д. Библиотека расположена в файле commdlg.dll на 16-разрядных Windows и comdlg32.dll на 32- и 64-разрядных. Ее относят к категории пользовательского интерфейса в API.

Библиотека общих элементов управления (Common Control Library)

Обеспечивает приложениям доступ к некоторым дополнительным элементам управления, предоставляемым операционной системой. Сюда относятся статусбары, прогрессбары, тулбары и вкладки. Библиотека располагается в файле commctrl.dll на 16-разрядных Windows и comctl32.dll на 32- и 64-разрядных. Ее относят к категории пользовательского интерфейса в API.

Оболочка Windows (Windows Shell)

Компонент Windows API, предоставляющий приложениям доступ к функциям оболочки операционной системы, а также возможности по ее изменению и улучшению. Этот компонент располагается в shell.dll на 16-разрядных Windows, и shell32.dll - на 32-разрядных и 64-разрядных. Легковесные вспомогательные функции оболочки (Shell Lightweight Utility Functions) находятся в shlwapi.dll и обычно относятся к категории пользовательского интерфейса в API.

Сетевые сервисы (Network Services)

Предоставляет доступ к различным сетевым возможностям операционной системы. Подкомпоненты данной категории включают NetBIOS, WinSock, NetDDE, удаленный вызов процедур (RPC) и многие другие. Этот компонент расположен в netapi32.dll на 32- и 64-разрядных Windows.

Особенности программирования с использованием WinAPI

Хранение строк

Как уже было сказано, большинство функций WinAPI, принимающих строки, имеют -A и -W версии. Однако для сохранения совместимости с до-NT приложениями в

заголовочных файлах объявлены версии этих функций без суффиксов, которые транслируются в конкретную версию в зависимости от флагов сборки. Этими флагами являются `UNICODE` и `_UNICODE`. При работе в MSVS есть возможность задать их из настроек проекта, в противном случае для поддержки Юникода нужно объявить оба флага самостоятельно. При использовании Юникода все символы должны иметь тип `WCHAR`, наоборот - `CHAR`. Строковые константы, соответственно, будут записаны как `L"value"` и `"value"`.

TL;DR

Чтобы не иметь проблем с не-ASCII строками в программе (особенно это касается констант и традиции MSVS по умолчанию сохранять файлы в кодировке текущего пользователя - у нас CP-1251) рекомендуется использовать варианты функций без суффиксов кодировки, для хранения строк и символов использовать тип `TCHAR`, для записи строковых констант должен применяться макрос `_T("value")`, он же `T()` или `TEXT()`. В случае проблем с кодировкой исходника (привет, MSVS!) это поможет быстро переключать тип строк в проекте. Для доступа к соответствующей функциональности нужно сделать

```
#include <tchar.h>
```

Особенности наименования

В целом нужно добавить, что все типы данных, константы, модификаторы функций WinAPI объявлены прописными символами; некоторые из объявлений перекрывают ключевые слова языка Си: `VOID`, `CONST`, `CHAR` и т.д. Это всего-лишь указание на то, что интерфейс проектировался на заре существования ANSI C, если не раньше, и задумывался как широко портируемый как между платформами, так и между языками.

Венгерская нотация

Ещё одна особенность WinAPI (но не анахронизм) - именование полей структур и параметров функций в венгерской нотации. Суть её в том, что в префиксе каждого названия содержится информация о типе поля. Например:

```
szWinClass:
    sz - строка, завершающаяся символом 0
```

или

```
lpApplicationName:
    lp - long pointer - указатель
```

Иногда подобная нотация использовалась и для сложных типов данных, например:

```
LPCWSTR:
    LP - указатель
    C - const
    WSTR - wide string - строка в Юникоде, т.е. из WCHAR'ов
```

Внезапно, подобная запись упрощает чтение и написание исходного кода - глаза не спотыкаются о `const`, `*`, `* const *` `type`, ...

Доступ к ресурсам

Опять же, ввиду значительного возраста WinAPI, он не является объектно-ориентированным в привычном плане, однако сами объекты там есть, и представлены они т.н. дескрипторами (англ. handle) - некоторая ссылка на конкретный ресурс, не обязательно указатель. Соответственно, разработчик может создавать или получать такие объекты специальными функциями, использовать их косвенно (путём передачи дескрипторов в другие специальные функции) и уничтожать по мере надобности (да, тоже с помощью отдельных функций).

Пара слов о стандартах

MisroSoft, пользуясь своим монопольным положением на рынке, весьма вольно относится к реализации стандартов языков программирования. Поддержка языка `C` в ее компиляторе (даже в самых последних версиях) осталась на уровне стандарта 1989 (!) года (ни один стандарт `C++` также не реализован полностью, но с этим в рамках лабораторных работ столкнуться, скорее всего, не придется). В связи с этим могут возникнуть трудности при попытке писать на `C` в MS VS в современном стиле. Необходимо проследить, чтобы компиляция кода производилась в режиме `C++` - для этого необходимо в качестве расширения файла с исходным кодом при его создании указать `.cpp`.

Описание примера

Зависимости

- user32[.lib]
- gdi32[.lib]
- comctl32[.lib]
- kernel32[.lib]

Этот список в теории не понадобится, поскольку создавать проект будем из MSVS, где уже есть соответствующий шаблон со всем нужным. Хотя для общего развития полезно создать пустой проект и добавить всё туда руками.

Почти все необходимые определения выполнены в одном заголовочном файле, поэтому будет достаточно сделать

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

Точка входа

Для графических приложений WinAPI предлагает специальную точку входа WinMain следующей сигнатуры:

```
int WINAPI WinMain (HINSTANCE hThisInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpszArgument,
                    int nCmdShow)
```

По сравнению с традиционной для C/C++ функцией main здесь будет ровно на 1.5 меньше телодвижений:

1. Не придётся получать дескриптор модуля, из которого был загружен исполняемый код текущего процесса (hThisInstance).
2. Не нужно руками указывать тип отображения окна при создании (nCmdShow).

Соответственно, если мы пользуемся стандартной точкой входа, необходимо сделать:

1. Получить дескриптор текущего модуля: `HINSTANCE hInstance = GetModuleHandle(NULL);` не забыв проверить на NULL.
2. В дальнейшем при создании окна в `ShowWindow()` использовать `SW_SHOW` вторым параметром - понятно, почему.

Создание окна

Перед созданием своего окна необходимо сообщить системе его свойства - создать класс этого окна и зарегистрировать его. Для первой задачи используется структура `WNDCLASS[EX]`. В данной структуре задается дескриптор модуля, которому будут соответствовать все окна данного класса, имя класса (должно быть уникальным!), указатель на процедуру-обработчик потока сообщений, предназначенных экземплярам окна этого класса (пока пишем `DefWindowProc` и ждём следующего раздела) и ещё кучу полезных и не очень вещей - ту же кисть для заливки (например, `COLOR_BACKGROUND`). Вариант с префиксом аналогичен тому, что и без, только имеет несколько дополнительных полей (дескриптор маленькой иконки и т.п.).

Далее созданный класс необходимо зарегистрировать в системе с помощью `RegisterClass[Ex]()`, не забыв проверить ненулевой результат. Здесь не рассматриваем особенности регистрации классов (внутри приложения, внутри системы, ...). Зарегистрировав класс в начале работы приложения, необходимо не забыть отозвать его регистрацию по окончании - для этого служит функция `UnregisterClass[Ex]()`. Обратите внимание, что она принимает *имя* класса, а не указатель или дескриптор на него!

После этого можно создать само окно с помощью `CreateWindow[Ex]()`, что вернёт нам дескриптор окна. Далее осталось лишь отобразить его через `ShowWindow()` и запустить цикл обработки сообщений (см. следующий раздел). Без этого цикла можно убедиться в создании окна, организовав задержку - например, через `getchar()`, если приложение консольное. Созданное окно удаляется при помощи `DestroyWindow()`.

Освобождение ресурсов традиционно происходит в обратном порядке:

1. Уничтожаем окошко.
2. Отзываем класс (если сами его создали и зарегистрировали).

RegisterClass

Регистрирует класс окна по его описанию.

Параметр - указатель на структуру описания класса.

Возвращает не-0 при успехе и 0 в случае неудачи.

UnregisterClass

Обратное действие к предыдущему.

Параметр - *имя* класса, а не указатель на структуру, обратите внимание!

CreateWindow

Создаёт окно заданного класса с заданными параметрами. Возвращает дескриптор созданного окна.

Параметры:

- Флаг расширенных возможностей. Не понадобится, пишем 0.
- Имя класса, к которому должно принадлежать наше окно. У нас где-то наверху объявлена соответствующая указываем, пишем её.
- Заголовок окна - строковая константа.
- Тип окна. У нас будет окно с возможностью изменения размера - `WS_OVERLAPPEDWINDOW`.
- Место отображения окна по горизонтали. Здесь и ниже указываем `CW_USEDEFAULT` - пусть ОС сама решает, где размещать окно.
- Место отображения окна по вертикали.
- Ширина окна в пикселях. Выставить нужное значение.
- Высота окна в пикселях. Выставить нужное значение.
- Дескриптор родительского окна. У нас окно вполне себе самостоятельное и не имеет родителя, но такого не бывает - у любого окна должен быть родитель, если он не нужен - пишем `HWND_DESKTOP`, и тогда родителем будет окно рабочего стола.
- Дескриптор контекстного меню. У нас его нет - `NULL`.
- Дескриптор экземпляра программы. Объявлен где-то в начале функции `main()` либо получен автоматически в качестве аргумента `WinMain()`.
- Дополнительные данные. Нам не нужны, будет `NULL`.

DestroyWindow

Уничтожает ранее созданное окно по его дескриптору.

ShowWindow

Управляет отображением окна с учётом требуемого типа отображения.

Параметры:

- Дескриптор окна.
- Команда отображения. Возможны варианты вроде "переместить на задний план", "свернуть", "показать неактивным", "показать активным". Иногда выбор действия осуществляется ещё до запуска нашего приложения - тогда команда приходит через аргумент точки входа в приложение. Соответствующий аргумент у нас назван `nCmdShow`, его и пишем.

Организация обработки сообщений

Для того чтобы созданное нами окно не закрывалось сразу после появления и начало реагировать на всевозможные события вроде изменения положения, перетягивания и т.д. необходимо реализовать цикл обработки сообщений. Суть в целом примерно следующая:

1. Получаем сообщение от ОС. При появлении сообщения о закрытии окна прерываем цикл.
2. Транслируем его из виртуальных кодов в символьные - рудимент старых версий Windows.
3. Передаём сообщение обработчику, определённому при описании класса окна.
4. Возвращаемся к п. 1.

Этих действий достаточно для привнесения в наше окно минимально-достаточной интерактивности, однако не всё так просто. Напомним, что при описании структуры класса окна мы указали в качестве обработчика сообщений `DefWindowProc`. Это стандартный обработчик сообщений, который обрабатывает минимально необходимый перечень типов сообщений (о закрытии, об очистке фона окна при перерисовке и о нажатиях клавиш мыши/клавиатуры - если для этих событий установлены перехватчики).

Сигнатура обработчика следующая:

```
LRESULT CALLBACK WindowProcedure(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
```

Где:

- `CALLBACK` - соглашение о вызовах, которому придерживается эта функция.
- `wParam`, `lParam` - параметры сообщения.

Общий принцип проектирования своего обработчика сообщений:

- В зависимости от типа сообщения осуществляем разбор его параметров и выполняем некоторый код-обработчик для данного сообщения. При этом если сообщение успешно обработано, то функция должна вернуть `0`.
- Если нам сообщение не интересно либо мы не смогли его нормально обработать, то передаём его стандартному обработчику `DefWindowProc()` и возвращаем то, что он вернул нам.
- Не забываем обработать сообщение об уничтожении окна (см. пример) путём добавления в очередь специального сообщения `WM_QUIT`.

Названия типов сообщений содержат префикс `WM_`, например, `WM_KEYDOWN`.

GetMessage

Блокирует поток программы, ожидая появления сообщения для конкретного окна.

Параметры:

- Указатель на переменную, в которую будет положено принятое сообщение. У нас есть такая переменная - `messages`, пишем указатель на неё.
- Дескриптор окна, сообщения кому следует ждать. Можно указать `NULL`, это добавит также сообщения текущему потоку, не адресованные ни одному окну.
- Фильтр сообщений по типу, нижняя граница. Каждое сообщение характеризуется типу (кодом), о чём будет сказано в дальнейшем. Здесь можно указать минимальное значение кода сообщения, которое будет принято. Нам фильтрация не нужна, поэтому пишем 0.
- Фильтр сообщений по типу, верхняя граница. Аналогично предыдущему.

Возвращает 1 в случае успеха, 0 в случае получения сообщения о выходе и -1 в случае ошибки.

TranslateMessage

Транслирует сообщение из виртуальных кодов в символьные. **TODO:** дописать.

Параметр один - указатель на сообщение, аналогично первому параметру предыдущей функции.

Функция не изменяет сообщение, переданное ей параметром, а создает новое и помещает его в очередь сообщений окна.

DispatchMessage

Передаёт сообщение обработчику, указанному в описании класса окна, которому предназначено это сообщение.

Параметр один и аналогичен предыдущей функции.

Смена цвета фона

Согласно заданию, необходимо выполнить смену цвета фона путём подмена кисти, используемой классом окна. Смотрите в сторону метода `SetClassLongPtr()`. Очевидно, перед этим надо создать новую кисть при помощи вызова `CreateSolidBrush()`, а потом не забыть её удалить методом `DeleteObject()`.

CreateSolidBrush

Функция принимает на вход цвет в формате RGB и возвращает дескриптор созданной кисти. Полученный дескриптор по окончании использования должен быть освобождён вызовом следующей функции.

Параметры:

- Цвет в формате RGB. Получить его можно с помощью макроса `RGB(red, green, blue)`, где параметры - целые числа от 0 до 255.

DeleteObject

Удаляет ранее созданный объект GDI по его дескриптору. В нашем случае будет использоваться для уничтожения кисти, используемой нашим классом окна.

Обработка событий клавиатуры

Для целей обработки клавиатуры в WinAPI предусмотрено несколько типов сообщений, как минимум `WM_KEYDOWN` и `WM_KEYUP`. Использовать первый из них для обработки *однократного* нажатия неудобно, поскольку сообщения такого типа генерируются всё время пока клавиша нажата. В отличие от них, сообщения второго типа генерируются ровно один раз в момент отпускания клавиши.

Для нормальной обработки сообщений недостаточно иметь информацию о его типе. Банально при обработке нажатой клавиши нужно знать *какая именно* клавиша была нажата. Здесь на помощь приходят дополнительные аргументы функции-обработчика сообщений. Названы они просто и почти понятно: `wParam` и `lParam`. Эти параметры могут содержать полезные для обработки сообщения данные, их интерпретация всегда зависит от типа сообщения и приведена в документации (MSDN). Конкретно для события `WM_KEYUP` будет достаточно `wParam` - он содержит код конкретной клавиши, которая была отпущена.

Обратим внимание на структуру функции-обработчика сообщений. Здесь мы осуществляем некоторые действия в зависимости от типа сообщения. При этом если мы их (действия) действительно выполнили - явно выходим из функции через `return 0;`. Это сделано для того, чтобы в случае, когда сообщение осталось нами не обработано (или в принципе не интересно), оно было передано стандартному обработчику `DefWindowProc()`, чей вызов должен осуществляться в самом конце нашей функции. Без этого нам бы пришлось вручную обрабатывать всё массу стандартных событий наподобие того же изменения размера и т.д. и т.п., тысячи их.

PostQuitMessage

Помещает в очередь сообщений новое типа `WM_QUIT`, предназначенное для текущего потока (*NB!* Именно поэтому второй параметр у `GetMessage()` указан как `NULL`). При его получении функция `GetMessage()` вернёт 0, тем самым завершив цикл обработки сообщений.

Параметр - код завершения приложения. Если указать не-0, ОС будет считать, что приложение завершило работу с ошибкой.

Обработка событий мыши

Обработка событий мыши осуществляется очень похожим образом, однако вместо передачи кода клавиши параметром для каждой кнопки мыши генерируются отдельные события на нажатие и отпускание. Их имена выглядят как `WM_xxxDOWN`, `WM_xxxUP` и `WM_xxxDBLCLK`, что означает "клавиша нажата", "клавиша отпущена" и "совершено двойное нажатие на клавишу" соответственно. `xxx` здесь подразумевает имя конкретной клавиши. Например, событие отпускания левой кнопки мыши кодируется константой `WM_LBUTTONUP`. Заметим, что параметры сообщений о состояниях клавиш мыши `wParam` и `lParam` также несут полезную информацию: первый содержит состояния клавиш-модификаторов (типа Ctrl и Shift), а второй - координаты x и y точки клика относительно левого верхнего угла окна, которые можно извлечь либо по отдельности при помощи макросов `GET_X_LPARAM/GET_Y_LPARAM`, либо сразу парой в структуру типа `POINTS` макросом `MAKEPOINTS`.

Для вычисления размера окна нам потребуется познакомиться с еще одной полезной функцией WinAPI: `GetClientRect`.

GetClientRect

Данная функция позволяет по дескриптору на окно получить его размер (а если быть более точным - координаты клиентской области окна) в структуру `RECT`. Левая и верхняя координата всегда равны 0, а правая и нижняя содержат ширину и высоту окна соответственно.

Поверхности для рисования, кисти и примитивы

Процесс рисования достаточно прост и логичен: берем холст, берем кисть (или создаем и берем новую), рисуем то, что хотели и сообщаем об окончании рисования. Здесь нам вновь пригодится теперь уже знакомая процедура `GetClientRect` для определения размеров окна (которые в данном случае будут совпадать с размерами поверхности рисования).

BeginPaint

Функция `BeginPaint` служит сразу двум целям: сообщает системе о начале обновления содержимого окна и получает холст, или *контекст рисования*, на котором, собственно, и будет выполняться отображение объектов. Ей на вход необходимо подать дескриптор окна, на котором мы хотим рисовать, а также указатель на выходной буфер параметров, которые могут понадобиться некоторым из функций рисования.

EndPaint

Процедура `EndPaint` сообщает системе о том, что рисование на контексте окна завершено, и можно отображать результат пользователю. До вызова этой функции на экране никаких изменений видно не будет.

Следует помнить, что `BeginPaint` и `EndPaint` всегда используются парой! Иначе можно напороться на странные и весьма трудноуловимые ошибки.

Итак, мы научились запрашивать и освобождать контекст рисования. Пришло время, наконец, что-нибудь нарисовать! Начнем с простого: заливки окна фоновым цветом.

FillRect

Окно приложения у нас, как правило, прямоугольное, поэтому для его заливки необходимо просто нарисовать закрашенный прямоугольник искомого размера. Именно это и делает функция `FillRect`. Она принимает три параметра: поверхность, на которой будет производиться рисование, координаты и размер области прямоугольника в виде структуры RECT, а также кисть, которой будет производиться отрисовка.

Внимательный читатель спросит: а где же мы возьмем эту кисть? Windows заботливо создала некоторый предопределенный набор кистей за нас во время запуска нашего приложения. Обратиться к ним мы можем, указав вместо дескриптора на кисть ее номер - специальные константы `COLOR_xxx`. Нас интересует кисть с цветом текущего фона окна (он зависит от системной темы), поэтому указываем константу `COLOR_WINDOW`.

Заметим, что необходимо добавить +1 к константе, поскольку в противном случае система не смогла бы отличить предопределенную кисть с номером 0 от недействительной кисти (`NULL`).

Ну а *самый внимательный* читатель заметит, что кисть, используемую для перерисовки фона окна, мы можем переопределить при старте нашего приложения, указав соответствующее поле в структуре `WNDCLASS`. Впрочем, обратиться к ней можно тем же самым способом: `COLOR_WINDOW+1`.

Однако, предварительно подготовленных кистей мало - в основном, они соответствуют стандартным цветам элементов интерфейса. Что делать, если нам нужно рисовать цветом, для которого готовой кисти нет? Естественно, создать свою! Для этого служит функция `CreatePen`.

CreatePen

Функция создает новую кисть с указанными параметрами линии (пунктир/сплошная/прерывистая), толщиной и цветом. Кисть - это ресурс, за которым необходимо следить и удалять после использования при помощи `DeleteObject`.

Многие функции рисования используют кисть, которая передана им параметром. Однако некоторые берут кисть по умолчанию, указанную в контексте рисования. Для того, чтобы переопределить эту кисть нашей, используем функцию `SelectObject`.

SelectObject

Задает для указанного первым параметром контекста рисования значение атрибута, указываемое вторым параметром. Необычной особенностью данной функции является то, что собственно атрибут выводится автоматически на основе типа переданного значения. Возвращает функция старое значение атрибута или `NULL`, если оно не было указано.

Обратим внимание, что кисть по умолчанию необходимо восстановить по окончании использования нашей собственной вторым вызовом метода `SelectObject` по окончании рисования.

Самая простая вещь (после сплошной заливки), которую можно нарисовать - это линия. Для этого необходимы две функции: `MoveToEx` и `LineTo`.

MoveToEx

Указывает текущую позицию на холсте (контексте рисования), позволяя, в том числе, получить предыдущую.

LineTo

Проводит линию кистью по умолчанию из текущей позиции на холсте до указанной.

Запуск других процессов

Для этого нам потребуется несколько структур:

- `STARTUPINFO` - хранит различные параметры запуска, такие как позиция и размеры окна, консоли (если консольное приложение), дескрипторы потоков ввода, вывода и ошибок, и т.д.
- Неиспользуемые поля должны быть обязательно заполнены нулями
- `PROCESS_INFORMATION` - после создания процесса будет хранить его дескриптор и т.п.
- Обратите внимание на раздел Remarks в описании этой структуры на MSDN!

После чего можно вызывать

```
CreateProcess(_T("path_to_exe"), _T("CLI args"),
              NULL, NULL, FALSE, 0, NULL, NULL,
              *STARTUPINFO, *PROCESS_INFORMATION)
```

не забыв проверить возвращаемое значение типа WINBOOL.

Полезные ссылки

- Пошаговое руководство по созданию приложений с использованием WinAPI https://msdn.microsoft.com/en-us/library/windows/desktop/ff381399(v=vs.85).aspx
- Организация цикла приёма сообщений https://msdn.microsoft.com/en-us/library/windows/desktop/ms644936(v=vs.85).aspx
- Типы данных в Windows https://msdn.microsoft.com/en-us/library/windows/desktop/aa383751(v=vs.85).aspx
- Исходный код ReactOS https://github.com/reactos/reactos/tree/master/reactos
- Документация по ReactOS https://doxygen.reactos.org
- Исходный код Windows Research Kernel https://github.com/markjandrews/wrk-v1.2
- Исходный код Wine https://github.com/wine-mirror/wine

Ссылки на реализацию функций

- GetModuleHandleA,W https://github.com/reactos/reactos/blob/master/reactos/dll/win32/kernel32/client/loader.c#L818
- RegisterClassA,W https://github.com/reactos/reactos/blob/master/reactos/win32ss/user/user32/windows/class.c#L1580
- CreateWindowExA,W https://github.com/reactos/reactos/blob/master/reactos/win32ss/user/user32/windows/window.c#L349
- ShowWindow https://github.com/reactos/reactos/blob/master/reactos/win32ss/user/user32/include/ntwrapper.h#L439
- DestroyWindow https://github.com/reactos/reactos/blob/master/reactos/win32ss/user/user32/include/ntwrapper.h#L386
- UnregisterClassA,W https://github.com/reactos/reactos/blob/master/reactos/win32ss/user/user32/windows/class.c#L1856

- PostQuitMessage
  https://github.com/reactos/reactos/blob/master/reactos/win32ss/user/user32/windows/message.c#L2300
- CreateSolidBrush
  https://github.com/reactos/reactos/blob/master/reactos/win32ss/gdi/gdi32/objects/brush.c#L196
- DeleteObject
  https://github.com/reactos/reactos/blob/master/reactos/win32ss/gdi/gdi32/objects/gdiobj.c#L315
- LoadIconA,W
  https://github.com/reactos/reactos/blob/master/reactos/win32ss/user/user32/windows/cursoricon.c#L2031
- LoadCursorA,W
  https://github.com/reactos/reactos/blob/master/reactos/win32ss/user/user32/windows/cursoricon.c#L2061
- GetMessageA,W
  https://github.com/reactos/reactos/blob/master/reactos/win32ss/user/user32/windows/message.c#L2066
- TranslateMessage[Ex]
  https://github.com/reactos/reactos/blob/master/reactos/win32ss/user/user32/windows/message.c#L2770
- DispatchMessage
  https://github.com/reactos/reactos/blob/master/reactos/win32ss/user/user32/windows/message.c#L1870
- CreateProcessA,W
  https://github.com/reactos/reactos/blob/master/reactos/dll/win32/kernel32/client/proc.c#L4768