Предисловие

Эта статья посвящена описанию программирования приложений на «чистом» Win32 API. Она написана в основном для начинающих программистов, пишущих программы на Visual C++ 6 с использованием библиотеки MFC, но я надеюсь, может пригодиться и более опытным людям.

First Blood

После создания нового проекта **Win32 Application,** в зависимости от выбранных опций, мастер генерирует стартовый код. Из этого кода программисту впоследствии, и придется писать программу. Создавая новый проект **Win32 Application,** выберите в окне мастера опцию **An empty project** и добавьте в раздел **Source Files** новый файл с расширением .cpp.

В этом файле добавьте функцию **WinMain** вида:

|  |
| --- |
| #include <windows.h>  int APIENTRY WinMain(HINSTANCE hInstance,  HINSTANCE hPrevInstance,  LPSTR lpCmdLine,  int nCmdShow)  {  MessageBox(NULL, "Простейшая программа!","WinAPI App", 0); //сообщение  return 0;  } |

Вот и готова первая программа на WinAPI. Она выводит сообщение, после чего завершает свою работу. Обратите внимание на параметры функции **WinMain**:

* **HINSTANCE hInstance** – дескриптор экземпляра приложения. Этот дескриптор содержит адрес начала кода программы в ее адресном пространстве. Дескриптор hInstanceчаще всего требуется функциям, работающим с ресурсами программы.
* **HINSTANCE hPrevInstance** – дескриптор предыдущего экземпляра приложения. Этот дескриптор остался от старых версий Windows - скорее всего, вам он никогда не пригодится.
* **LPSTR lpCmdLine** – указатель на начало командной строки, введенной при запуске программы.
* **int nCmdShow** – это значение содержит желаемый вид окна (например, свернутый или развернутый)

Значение, которое возвращается функцией **WinMain** (тип **int**) – код завершения программы. Принято, что если программа завершила свое выполнение без ошибок, возвращается 0.

Функция **WinMain** – первая функция, которая выполнятся в программе (ее еще называют **«точка входа»** или «entry point»). С нее все начинается, и ею (желательно) все должно закончиться.

|  |
| --- |
| **ПРИМЕЧАНИЕ**  Функция WinMain – это первая функция, которую вы можете увидеть и заполнить кодом. На самом деле до этой функции выполняется достаточно много кода из библиотеки C++ |

You have a Message!

Программисты, незнакомые с программированием на WinAPI, спросят: «Что с этим делать?!», - или: «Где создавать CDialog?». В данном случае ответ прост – нигде! В нашем проекте нет класса CDialog или, предположим, CButton – ведь эта статья посвящена тому, как обойтись без них.

В Windows при каждом событии, произошедшем в системе, отсылается **«сообщение Windows»** («windows message»). Эти сообщения уведомляют программу о событиях в системе, а программа в свою очередь, может на них реагировать. Сообщения может отсылать не только Windows, но и сами приложения. Это является одним из способов организации связи между процессами в системе. Конечно, программа может отсылать сообщения и самой себе.

|  |
| --- |
| **СОВЕТ**  Сообщение можно отослать функцией SendMessage или ее асинхронным аналогом PostMessage. |

Для приема сообщений в программе должен находиться **«цикл сообщений»** («message loop») который обычно выглядит так:

|  |
| --- |
| //цикл сообщений приложения  MSG msg = {0}; //структура сообщения  int iGetOk = 0; //переменная состояния  while ((iGetOk = GetMessage(&msg, NULL, 0, 0 )) != 0) //цикл сообщений  {  //если GetMessage вернул ошибку - выход  if (iGetOk == -1) return 3;  TranslateMessage(&msg);  DispatchMessage(&msg);  } |

Функция **GetMessage** принимает следующие параметры:

* **LPMSG lpMsg** – указатель на структуру сообщения, в которую **GetMessage** вернет результат.
* **HWND hWnd** – описатель окна, от которого **GetMessage** примет сообщение (**NULL** означает, что **GetMessage** принимает сообщения от всех окон, принадлежащих потоку).
* **UINT wMsgFilterMin** – наименьший идентификатор сообщения, которое примет **GetMessage.**
* **UINT wMsgFilterMax** – наибольший идентификатор сообщения, которое примет **GetMessage** (если в значениях параметров **wMsgFilterMin** и **wMsgFilterMax** передать 0, функция будет принимать **ВСЕ** сообщения).

Функция **GetMessage** не отдает управление программе, пока не придет какое-либо сообщение. Если пришедшее сообщение – **WM\_QUIT**, функция **GetMessage** вернет **0**. Тогда цикл прервется, и программа завершит свою работу. При любом другом сообщении функция **GetMessage** возвращает значение больше нуля, и начинатся выполнение тела цикла. При ошибке **GetMessage** возвращает -1.

|  |
| --- |
| **СОВЕТ**  Сообщение WM\_QUIT лучше посылать с помощью специальной функции PostQuitMessage(int iExitCode). Эта функция отошлет сообщение WM\_QUIT, а в параметре wParam передаст код завершения программы, указанный в iExitCode. |

Функция **DispatchMessage** должна вызвать «функцию обработки сообщений». В простейшем варианте она выглядит так:

|  |
| --- |
| LRESULT CALLBACK WndProc(HWND hWnd, UINT message,  WPARAM wParam, LPARAM lParam)  {  // выборка и обработка сообщений  switch (message)  {  case WM\_LBUTTONUP:  //реакция на сообщение  MessageBox(hWnd,"Вы кликнули!","событие",0);  break;  case WM\_DESTROY:  //реакция на сообщение  PostQuitMessage(0);  break;  //все необработанные сообщения обработает сама Windows  default:  return DefWindowProc(hWnd, message, wParam, lParam);  // switch (message)  }  return 0;  } // конец функции обработчика сообщений |

При вызове этой функции ей передаются следующие параметры:

* **HWND hWnd** – описатель окна, от которого пришло сообщение.
* **UINT message** – идентификатор сообщения.
* **WPARAM wParam** и **LPARAM lParam** – параметры сообщения.

Функция обработки сообщений не обязательно должна иметь имя **WndProc**. Таких функций в программе может быть несколько, но их прототипы обязательно должны выглядеть так:

|  |
| --- |
| LRESULT CALLBACK WndProc(HWND hWnd, UINT message,  WPARAM wParam, LPARAM lParam) |

При вызове этой функции **DispatchMessage** передает в параметре **message** идентификатор сообщения. По этому идентификатору производится выборка и выполняется какое-либо действие (**«реакция на сообщение»**).

В Windows существует очень много сообщений! Писать обработчики для всех сообщений – нереальная задача. Чтобы Windows сама обработала бесполезное для вас сообщение, необходимо вызвать функцию **DefWindowProc**:

|  |
| --- |
| LRESULT DefWindowProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam) |

Желательно передавать все необработанные сообщения этой функции, а результат ее выполнения возвращать при выходе из **WndProc**. Это очень важно, так как от обработки некоторых сообщений Windows ждет возврата конкретных результатов или действий.

Одной функцией обработки сообщений могут пользоваться несколько окон, но для **одного** окна может существовать только **одна** функция обработки сообщений! Как же система определяет, какой именно функцией обработки сообщения пользоваться для конкретного окна и где она находится?! За это отвечает **«класс окна»** («window class»).

CLASSные окна

При создании нового окна ему присваивается **«Класс окна»** (window class). Класс окна задает оконную функцию, используемую по умолчанию. Кроме этого, класс окна задает другие параметры окна, такие, как стиль, меню окна, цвет рабочей области и т.д. Разные классы окон могут указывать на одну и ту же функцию обработки сообщений. Для создания класса его необходимо зарегистрировать.

Итак, регистрация! За нее отвечает функция **RegisterClass**. В ее параметре необходимо передать указатель на структуру **WNDCLASS**. Обычно для заполнения структуры и вызова **RegisterClass** создают отдельную функцию. Но это - дело вкуса.

Вот простейший пример такой функции:

|  |
| --- |
| ATOM RegMyWindowClass(HINSTANCE hInst, LPSTR lpzClassName)  {  WNDCLASS wcWindowClass = {0};  //адрес ф-ции обработки сообщений  wcWindowClass.lpfnWndProc = (WNDPROC)WndProc;  //стиль окна  wcWindowClass.style = CS\_HREDRAW|CS\_VREDRAW;  //дискриптор экземпляра приложения  //название класса  wcWindowClass.hInstance = hInst;  wcWindowClass.lpszClassName = lpzClassName;  //загрузка курсора  wcWindowClass.hCursor = LoadCursor(NULL, IDC\_ARROW);  //загрузка цвета окон  wcWindowClass.hbrBackground = (HBRUSH)COLOR\_APPWORKSPACE;  //регистрация класса  return RegisterClass(&wcWindowClass);  } |

* **WNDPROC lpfnWndProc** – адрес функции обработки сообщений.
* **HINSTANCE hInstance** – уже знакомая переменная, описывающая экземпляр.
* **LPCTSTR lpszClassName** – имя нового класса.
* **HICON hCursor** – описатель курсора мыши.
* **HBRUSH hbrBackground** – цвет рабочей области окна.

Функция **RegisterClass** возвращает уникальный «описатель класса окна» типа **ATOM**. Если при регистрации класса произошла ошибка, это значение будет равно нулю. Чтобы узнать, что произошло, можно вызвать функцию **GetLastError()**.

Существует также функция RegisterClassEx. Это аналог функции **RegisterClass** с возможностью присвоения окнам маленькой иконки. При работе с этой функцией необходимо пользоваться структурой **WNDCLASSEX.**

|  |
| --- |
| **ПРИМЕЧАНИЕ**  Если вы решились работать с GUI Windows вручную, то пользоваться нужно именно RegisterClassEx, поскольку приложение, не имеющее маленькой иконки, сегодня выглядит в Windows как минимум странно. – прим.ред. |

|  |
| --- |
| **СОВЕТ**  Следите, чтобы имя вашего класса не совпадало с именами системных классов (например: button или edit). |

|  |
| --- |
| **ПРИМЕЧАНИЕ**  Я описал не всю структуру. Все незаполненные поля, которых нет в примере, сейчас равны нулю. Об их значениях можно узнать из MSDN. |

Сообщения от окон, созданных на базе класса, зарегистрированного описанной выше функцией RegMyWindowClass, будут обрабатываться функцией с именем WndProc. Чтобы функция WndProc поняла, от какого именно окна пришло сообщение, ей передается уникальный описатель окна **HWND**.

Our Windows

На вашем месте у меня возникло бы желание увидеть те самые пресловутые окна, из-за которых столько шума. Окно в Windows создается функцией **CreateWindow**. Вот ее прототип:

|  |
| --- |
| HWND CreateWindow(LPCTSTR lpClassName,  LPCTSTR lpWindowName,  DWORD wStyle,  int x,  int y,  int nWidth,  int nHeight,  HWND hWndParent,  HMENU hMenu,  HINSTANCE hInstance,  LPVOID lpParam); |

Как видите, у функции множество параметров:

* **LPCTSTR lpClassName –** имя класса для создаваемого окна (это имя использовалось при регистрации класса).
* LPCTSTR lpWindowName – имя окна.
* **DWORD dwStyle** – стиль окна.
* **int x** – позиция по горизонтали верхнего левого угла окна.
* **int y** – позиция по вертикали.
* **int nWidth** – ширина окна.
* **int nHeight** – высота окна.
* **HWND hWndParent** – используется для создания **«дочернего окна»** («child window»). Сюда передается описатель **«родительского окна»** («parent window»).
* **HMENU hMenu** – описатель меню (если **hMenu** равно нулю, используется меню класса, указанного в **lpClassName**).
* **HINSTANCE hInstance** – экземпляр приложения.
* **LPVOID lpParam** – указатель на пользовательский параметр окна. Этот указатель со всеми остальными параметрами функции **CreateWindow** будет занесен в структуру **CREATESTRUCT**. В сообщениях **WM\_CREATE** или **WM\_NCCREATE** параметр **lParam** будет содержать указатель на эту структуру.

Функция **CreateWindow** возвращает уникальный описатель окна **HWND**. Если функция вернула ноль, значит, во время создания окна произошла ошибка. Какая именно, можно узнать, вызвав функцию **GetLastError**.

|  |
| --- |
| **ПРИМЕЧАНИЕ**  Существует также функция CreateWindowEx, в которой дополнительно присутствует параметр dwExStyle. С его помощью можно создать окно с дополнительными стилями. |

План полета

Итак, сейчас я упрощенно расскажу, что же произойдет, если щелкнуть по окну левой кнопкой мыши.

1. Пользователь нажимает левую кнопку мыши в то время когда курсор мыши находится над рабочей областью окна.
2. Windows помещает сообщение **WM\_LBUTTONDOWN** в очередь потока.
3. Цикл обработки сообщения должен вынуть сообщение с помощью функции **GetMessage** и передать его на обработку функции **DispatchMessage**.
4. Функция **DispatchMessage** находит окно, которому предназначено сообщение и помещает сообщение в его очередь.
5. Функция окна обрабатывает сообщение **WM\_LBUTTONDOWN** и возвращает результат.
6. Тело цикла заканчивается, и управление снова передается функции **GetMessage** для ожидания новых сообщений.

Итого

WinMain, регистрация класса, цикл сообщений, функция обработки сообщений, создание окна... Как все это связать?! Вот код, который объединяет все написанное выше в одну программу:

|  |
| --- |
| #include <windows.h>  // объявление функций  LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);  ATOM RegMyWindowClass(HINSTANCE, LPCTSTR);  //////////////////////////////////////////////////////////////////////////  // функция вхождений программы WinMain  int APIENTRY WinMain(HINSTANCE hInstance,  HINSTANCE hPrevInstance,  LPSTR lpCmdLine,  int nCmdShow)  {  // имя будущего класса  LPCTSTR lpzClass = TEXT("My Window Class!");  // регистрация класса  if (!RegMyWindowClass(hInstance, lpzClass))  return 1;  // вычисление координат центра экрана  RECT screen\_rect;  GetWindowRect(GetDesktopWindow(),&screen\_rect); // разрешение экрана  int x = screen\_rect.right / 2 - 150;  int y = screen\_rect.bottom / 2 - 75;  // создание диалогового окна  HWND hWnd = CreateWindow(lpzClass, TEXT("Dialog Window"),  WS\_OVERLAPPEDWINDOW | WS\_VISIBLE, x, y, 300, 150, NULL, NULL,  hInstance, NULL);  // если окно не создано, описатель будет равен 0  if(!hWnd) return 2;  // цикл сообщений приложения  MSG msg = {0}; // структура сообщения  int iGetOk = 0; // переменная состояния  while ((iGetOk = GetMessage(&msg, NULL, 0, 0 )) != 0) // цикл сообщений  {  if (iGetOk == -1) return 3; // если GetMessage вернул ошибку - выход  TranslateMessage(&msg);  DispatchMessage(&msg);  }  return msg.wParam; // возвращаем код завершения программы  }  //////////////////////////////////////////////////////////////////////////  // функция регистрации класса окон  ATOM RegMyWindowClass(HINSTANCE hInst, LPCTSTR lpzClassName)  {  WNDCLASS wcWindowClass = {0};  // адрес ф-ции обработки сообщений  wcWindowClass.lpfnWndProc = (WNDPROC)WndProc;  // стиль окна  wcWindowClass.style = CS\_HREDRAW|CS\_VREDRAW;  // дискриптор экземпляра приложения  wcWindowClass.hInstance = hInst;  // название класса  wcWindowClass.lpszClassName = lpzClassName;  // загрузка курсора  wcWindowClass.hCursor = LoadCursor(NULL, IDC\_ARROW);  // загрузка цвета окон  wcWindowClass.hbrBackground = (HBRUSH)COLOR\_APPWORKSPACE;  return RegisterClass(&wcWindowClass); // регистрация класса  }  //////////////////////////////////////////////////////////////////////////  // функция обработки сообщений  LRESULT CALLBACK WndProc(  HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)  {  // выборка и обработка сообщений  switch (message)  {  case WM\_LBUTTONUP:  // реакция на сообщение  MessageBox(hWnd, TEXT("Вы кликнули!"), TEXT("событие"), 0);  break;  case WM\_DESTROY:  PostQuitMessage(0); // реакция на сообщение  break;  default:  // все сообщения не обработанные Вами обработает сама Windows  return DefWindowProc(hWnd, message, wParam, lParam);  }  return 0;  } |

Вот, в принципе, и все! Это полноценное приложение на WinAPI.

Программа регистрирует класс, создает окно этого класса и обслуживает сообщение **WM\_LBUTTONUP** (оно приходит по событию отпускания левой кнопки мыши), показывает окно, и после обработки сообщения снова возвращается в цикл сообщений, находящийся в **WinMain**.

PS

Признаю, что данная статья и программа опускает очень много деталей! Многие вещи были не раскрыты (например, остальные переменные структуры **WNDCLASS**). Все это сделано для того, чтобы максимально упростить статью и уменьшить код программы.

Буду рад получить отзывы и критику.