https://nestor.minsk.by/kg/2008/19/kg81911.html

[https://nestor.minsk.by/kg/logo.jpg](https://nestor.minsk.by/kg/)

[](https://nestorclub.com/index.pl?act=NEWSSHOW&id=2020120301)

**Ассемблер под Windows для чайников. Часть 3**

[**главная**](https://nestor.minsk.by/kg/)**»**[**статьи**](https://nestor.minsk.by/kg/arch.html)**»**[**2008**](https://nestor.minsk.by/kg/2008/)**»**[**19**](https://nestor.minsk.by/kg/2008/19/)**» 15.05.2008‣программирование**

**Сегодня мы научимся создавать окно. Окно — это достаточно сложная, но в то же время очень важная штуковина в Windows. Windows (от англ.: окна) потому так и называется, что большинство операций в ней производится с окнами программ. Так удобнее всего работать с множеством приложений.**

Может быть, когда-нибудь на смену окнам придут кубики или шарики, но на современных компьютерах пользователю удобнее всего работать с окнами. Вывод: для создания удобной в использовании программы нам надо научиться создавать окна. Этим мы сегодня и займемся. Сегодняшний урок будет заметно сложнее предыдущих, но я верю, что вы справитесь. Запускаем компилятор FASMW и вводим следующий код программы:

**format PE GUI 4.0  
entry start  
  
include 'win32a.inc'  
  
section '.data' data readable writeable  
  
\_class db 'FASMWIN32',0  
\_title db 'Пустое Окно',0  
\_error db 'Ошибка',0  
  
wc WNDCLASS 0,WindowProc,0,0,0,0,0,COLOR\_BTNFACE+1,0,\_class  
  
msg MSG  
  
section '.code' code readable executable  
  
start:  
  
invoke GetModuleHandle,0  
mov [wc.hInstance],eax  
invoke LoadIcon,0,IDI\_APPLICATION  
mov [wc.hIcon],eax  
invoke LoadCursor,0,IDC\_ARROW  
mov [wc.hCursor],eax  
invoke RegisterClass,wc  
cmp eax,0  
je error  
  
invoke CreateWindowEx,0,\_class,\_title,WS\_VISIBLE+WS\_DLGFRAME+   
WS\_SYSMENU,128,128,256,192,0,0,[wc.hInstance],0  
cmp eax,0  
je error  
msg\_loop:  
invoke GetMessage,msg,0,0,0  
cmp eax,0  
je end\_loop  
invoke TranslateMessage,msg  
invoke DispatchMessage,msg  
jmp msg\_loop  
  
error:  
invoke MessageBox,0,\_error,0,MB\_ICONERROR+MB\_OK  
  
end\_loop:  
invoke ExitProcess,[msg.wParam]  
  
proc WindowProc hwnd,wmsg,wparam,lparam  
push ebx esi edi  
cmp [wmsg],WM\_DESTROY  
je .wmdestroy  
.defwndproc:  
invoke DefWindowProc,[hwnd],[wmsg],[wparam],[lparam]  
jmp .finish  
.wmdestroy:  
invoke PostQuitMessage,0  
mov eax,0  
.finish:  
pop edi esi ebx  
ret  
endp  
  
section '.idata' import data readable writeable  
  
library kernel32,'KERNEL32.DLL',\  
user32,'USER32.DLL'  
  
include 'api\kernel32.inc'  
include 'api\user32.inc'**

Столько кода, и ничего особенного в результате. В MessageBox хотя бы присутствовала кнопка ОК, а тут — пустое окно. Не волнуйтесь: как только мы разберемся с устройством простейшего окна, мы сразу же перейдем к навешиванию на него прочих элементов. Используя MessageBox, мы были ужасно ограничены набором стандартных параметров этой функции. Кроме того, программа не могла продолжать исполняться до тех пор, пока пользователь не прореагирует нажатием кнопки в окошке или его закрытием. В случае использования окна программа может исполняться, параллельно реагируя на действия пользователя. Сразу будет нелегко понять алгоритм работы окна, но, как только вы его поймете, все остальное пойдет намного легче. Приступаем к разбору полетов. В секции данных после объявления трех текстовых строк (класс окна, его заголовок, сообщение об ошибке) объявляются две структуры данных. Первая — WNDCLASS — структура класса окна. Она описана в файле FASM\INCLUDE\EQUATES\USER32.INC (смотрим файл прямо в компиляторе или через блокнот — кому как удобно) и содержит 10 атрибутов класса окна: style — стиль окна; lpfnWndProc — указатель на процедуру обработки сообщений, посланных окну; cbClsExtra — количество дополнительных байт в памяти для данной структуры; cbWndExtra — количество дополнительных байт в памяти для данных, присоединенных к окну; hInstance — идентификатор приложения, создавшего окно; hIcon — идентификатор иконки окна; hCursor — идентификатор курсора окна; hbrBackground — цвет фона окна; lpszMenuName — указатель на имя меню окна; указатель на имя класса. Вторая — MSG — структура сообщения. Описана там же, где и первая. Состоит из шести элементов: hwnd — идентификатор окна — получателя сообщения; message — сообщение; wParam — дополнительная информация о сообщении; lParam — дополнительная информация о сообщении; time — время отправки сообщения; pt — координаты курсора на время отправки сообщения. Последний элемент, кстати, тоже является структурой, которая описана в том же файле. Это наглядный пример вложенной структуры. Не старайтесь все сразу запомнить — просто примите к сведению. После объявления структур wc будет считаться указателем на первый байт структуры данных, созданной в соответствии с шаблоном WNDCLASS, а msg — указателем на MSG. Элементам структуры wc присваиваются соответствующие значения, разделенные запятыми (в основном пока что ноли). Значения элементов структуры msg считаются сейчас неопределенными — хотя на практике они будут иметь нулевые значения, правильно считать их значения неопределенными. В тексте программы можно обращаться к отдельным элементам структуры, например, wc.hCursor или wc.hIcon, хотя в оперативной памяти все данные будут просто идти друг за другом безо всяких пометок. Элементы структур для 32-битных версий windows обычно имеют размер 32 бита — это 4 байта или двойное слово (dword). Поэтому в описании структуры написано dd (data in dwords). Следовательно, мы могли бы заменить, например, wc.hInstance на wc+16, потому что элементы структуры wc в адресном пространстве будут выглядеть примерно так:

Элемент Адрес  
wc.style wc+0  
wc.lpfnWndProc wc+4  
wc.cbClsExtra wc+8  
wc.cbWndExtra wc+12  
wc.hInstance wc+16  
wc.hIcon wc+20  
wc.hCursor wc+24  
wc.hbrBackground wc+28  
wc.lpszMenuName wc+32  
wc.lpszClassName wc+36



Можно было бы сейчас не вдаваться в эти подробности, но лучше, если вы сразу получите правильное представление о размещении данных в памяти. Теперь попробуем разобраться с исполняемым кодом (section '.code'). Функция GetModuleHandle возвращает в eax идентификатор исполняемого модуля. Она имеет всего один параметр — указатель на строку — имя модуля (exe или dll). Если параметр равен нулю, то функция возвращает идентификатор вызвавшего функцию модуля, то есть нашей программы. Этот идентификатор нужен нам для заполнения элемента wc.hInstance, поэтому следующей командой мы сразу помещаем в этот элемент содержимое eax. LoadIcon загружает указанную иконку из ресурсов исполняемого файла. Первый параметр — идентификатор исполняемого файла или ноль для загрузки стандартной иконки. Второй параметр — строка — имя иконки из ресурсов либо идентификатор стандартной иконки, если первый параметр ноль. В нашем случае иконка стандартная. IDI\_APPLICATION — это константа — ее и идентификаторы других стандартных иконок вы легко отыщете все в том же EQUATES\USER32.INC. Возвращаемое значение — идентификатор загруженной иконки, который мы тут же помещаем в wc.hIcon. LoadCursor работает по аналогии. Идентификаторы стандартных курсоров вы найдете чуть выше идентификаторов иконок. Теперь, когда все необходимые данные структуры wc находятся на своих местах, вызывается функция RegisterClass. Единственный параметр этой функции — указатель на структуру, содержащую описание класса окна. Если класс был успешно зарегистрирован, возвращаемое значение будет отлично от нуля. Если по каким-либо причинам не удалось зарегистрировать класс, то в eax вернется ноль. Исходя из этого, мы сравниваем eax и 0 (cmp) и в случае равенства прыгаем на метку ошибки (je error). Если не равно нулю — значит, все в порядке, и мы переходим к созданию окна. Функция CreateWindowEx создает окно. Параметры соответственно: расширенный стиль окна; указатель на зарегистрированное имя класса; указатель на имя окна; стандартный стиль окна; X координата левого верхнего угла окна; Y координата левого верхнего угла окна; ширина окна; высота окна; идентификатор родительского окна или окна-владельца; идентификатор меню или дочернего окна; идентификатор исполняемого модуля, с которым связано окно; указатель на значение, которое передается окну через структуру CREATESTRUCT в параметре lParam сообщения WM\_CREATE. Возвращаемое значение — идентификатор созданного окна. В случае ошибки возвращается ноль. Названия расширенных стилей окна вы можете найти в файле EQUATES\USER32.INC в группе Extended Window Styles, названия стандартных стилей находятся чуть выше в группах Window Styles и Common Window Styles.Далее следует цикл msg\_loop. Этот цикл будет повторяться до тех пор, пока окно не будет закрыто. Функция GetMessage получает сообщение из очереди сообщений приложения. Если сообщения отсутствуют, функция ожидает их, и цикл приостанавливается до появления нового сообщения. Сообщения посылаются окну операционной системой, когда с окном происходит какое-либо действие — например, когда окно перемещается, изменяется его размер или даже когда курсор мыши просто движется над областью окна. Также окну могут передаваться сообщения от других процессов. Параметры функции следующие: указатель на структуру, в которой разместятся элементы сообщения; идентификатор окна — получателя сообщения: если ноль, то сообщения принимаются для любого окна данного приложения; минимальное значение сообщения; максимальное значение сообщения. Последние два параметра исполняют роль фильтра сообщений. Так как все сообщения являются целочисленными значениями, можно установить фильтрацию типа "от… и до…" Если максимальное и минимальное значения равняются нулю, фильтрация не выполняется, и принимаются все сообщения без исключения. Если функция получает сообщение WM\_QUIT (выход), то возвращает ноль. В других случаях eax не будет равен нулю. Следовательно, если eax равен нулю, мы выходим из цикла (je end\_loop), иначе цикл продолжается. TranslateMessage переводит комбинации wm\_KeyDown/Up в wm\_Char или wm\_DeadChar, а комбинации wm\_SysKeyDown/Up — в wm\_SysChar или wm\_SysDeadChar — и отправляет переведенное сообщение снова в очередь. Таким образом в процедуру обработки сообщений поступят и виртуальные клавиши и их символьные значения. DispatchMessage передает сообщения процедуре обработки сообщений (WindowProc). В газетной статье нет возможности привести список всех сообщений и их описание, поэтому я рекомендую вам скачать англоязычную справку по API-функциям размером около 22 Мб: [сайт](http://ghirai.com/hutch/files/win32hlp_big.zip) . К сожалению, нормального аналогичного справочника на русском языке я вам посоветовать не могу. Так уж сложилось, что не в моде у программистов русский язык.



Процедура обработки сообщений WindowProc вызывается каждый раз при получении окном нового сообщения и передаче его через DispatchMessage. Синтаксис записи этой и других процедур предельно прост: макроинструкция proc, имя процедуры и ее параметры через запятую, которые по умолчанию считаются 32-битными (4 байта каждый). Возврат из процедуры осуществляет команда ret. Каждая процедура должна завершаться инструкцией endp, которая указывает компилятору, где заканчивается код данной процедуры и начинается следующий фрагмент кода. Команда push помещает значение указанного регистра в стек, а команда pop извлекает последнее значение из стека в указанный регистр. Стек — это специально выделенная область памяти для передачи или сохранения данных. Его можно представить, например, как вертикальную штангу тренажера, на которую можно одевать по одному грузу и снимать тоже по одному. Команда push будто указывает, откуда взять груз, а команда pop сообщает, куда поместить груз, снятый со штанги. Такую аналогию я привел для того, чтобы вы уже таки уяснили, что, если мы хотим сохранить содержимое нескольких регистров в стек, а потом вернуть эти значения в регистры, то нам необходимо сохранять в одном порядке, а извлекать — в обратном. Это важно запомнить и не путать: если сохраняем push eax ebx ecx — извлекаем pop ecx ebx eax. А сохраняем мы содержимое этих регистров, чтобы предотвратить потерю данных: функция DefWindowProc спокойно может затереть содержимое этих регистров. Так как окно у нас простейшее, и на нем нет никаких кнопок, кроме крестика для закрытия, единственное сообщение, которое мы должны обработать — это WM\_DESTROY, которое будет послано окну перед его удалением. Стало быть, если сообщение равно WM\_DESTROY, мы прыгаем на метку .wmdestroy, где будет вызвана функция PostQuitMessage, параметром которой является код завершения. Ноль означает, что программа самостоятельно завершает работу. Эта функция посылает нашему процессу сообщение WM\_QUIT, после обработки которого функция GetMessage вернет ноль, и цикл обработки сообщений прервется переходом на end\_loop — выход из программы. Если сообщение не WM\_DESTROY, то выполняется следующая функция — DefWindowProc. Эта функция обычно вызывается после того, как обработаны все предусмотренные нами сообщения (в нашем случае предусмотрено только одно, но их ведь может быть и сотня), для того, чтобы операционная система своими средствами произвела стандартную обработку сообщения, которое получило наше окно. Параметры функции: идентификатор окна — получателя сообщения; сообщение; дополнительная информация о сообщении; дополнительная информация о сообщении. Что ж, теперь мы можем навесить на наше окно кнопку. Для этого добавим в секцию данных (section '.data') ее класс и имя, а заголовок главного окна немного подправим:

**\_title db 'НеПустое Окно',0  
\_classb db 'BUTTON',0  
\_textb db 'КНОПКА',0**  
В секцию кода добавим обработку сообщения WM\_CREATE, которое приходит окну один раз при его создании, чтобы при создании главного окна создавать дочерние окна. Кнопка — это окно стандартного класса BUTTON, поэтому регистрировать этот класс нам не понадобится. Также добавим обработку сообщения WM\_COMMAND, которое приходит окну, когда пользователь выбирает пункт меню или совершает действие с другим дочерним элементом окна (в нашем случае — нажатие на кнопку):

**proc WindowProc hwnd,wmsg,wparam,lparam  
push ebx esi edi  
cmp [wmsg],WM\_CREATE  
je .wmcreate  
cmp [wmsg],WM\_COMMAND  
je .wmcommand  
cmp [wmsg],WM\_DESTROY  
je .wmdestroy  
.defwndproc:  
invoke DefWindowProc,[hwnd],[wmsg],[wparam],[lparam]  
jmp .finish  
.wmcreate:  
invoke CreateWindowEx,0,\_classb,\_textb,WS\_VISIBLE+WS\_CHILD+BS\_PUSHBUTTON, 10,10,100,50,[hwnd],1001,[wc.hInstance],NULL  
jmp .finish  
.wmcommand:  
cmp [wparam],1001  
jne .finish  
invoke MessageBox,[hwnd],\_textb,\_title,0  
jmp .finish  
.wmdestroy:  
invoke PostQuitMessage,0  
mov eax,0  
.finish:  
pop edi esi ebx  
ret  
endp**

Теперь при создании окна выполнятся команды после метки .wmcreate, и на главном окне будет создана кнопка с идентификатором 1001. При нажатии на эту кнопку окно получает сообщение WM\_COMMAND, а wparam сообщения будет содержать в старших двух байтах BN\_CLICKED (кликнута кнопка), а в младших двух байтах — идентификатор кнопки (1001). Так как константа BN\_CLICKED равна нулю (убеждаемся в этом в EQUATES\USER32.INC), можно не учитывать ее и просто сравнить wparam с 1001, чтобы убедиться, что была нажата наша кнопка. Если не равно — значит, не наша или не нажата — jne .finish . Иначе — показываем мессадж-бокс. По аналогии можете самостоятельно добавить еще пару кнопок, только не забывайте изменять координаты их местоположения, иначе они будут накладываться друг на друга, и вы увидите лишь одну из них. Поэкспериментируйте со стилями и другими параметрами. К следующему занятию постарайтесь четко усвоить принципы создания и работы простых окон, чтобы мы смело могли двигаться дальше.

Все приводимые примеры были протестированы на правильность работы под Windows XP и, скорее всего, будут работать под другими версиями Windows, однако я не даю никаких гарантий их правильной работы на вашем компьютере. Исходные тексты программ вы можете найти на форуме: [сайт](http://forum.sa-sec.org/index.php?showtopic=766) .  
  
**BarMentaLisk, q@sa-sec.org**

*Компьютерная газета. Статья была опубликована в номере 19 за 2008 год в рубрике*[*программирование*](https://nestor.minsk.by/kg/abc/%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5/)