https://nestor.minsk.by/kg/2008/20/kg82019.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 4**

[**главная**](https://nestor.minsk.by/kg/)**»**[**статьи**](https://nestor.minsk.by/kg/arch.html)**»**[**2008**](https://nestor.minsk.by/kg/2008/)**»**[**20**](https://nestor.minsk.by/kg/2008/20/)**» 22.05.2008‣программирование**

**В прошлый раз мы научились создавать окно с кнопкой. Сегодня продолжим тему про окна: будем учиться навешивать на окно другие элементы. Заодно и разберем основы ввода и вывода данных в окнах. В большинстве программ необходимо обеспечивать ввод и вывод данных. То есть то, что вы набрали на клавиатуре, должно каким-то образом попадать в память для последующей обработки, а также данные из памяти необходимо выводить на экран.**

Windows предоставляет широчайший набор функций для ввода и вывода данных, однако мы рассмотрим один из самых простых и популярных методов: использование поля для редактирования (окно стандартного класса EDIT). Как обычно, подробности будут ниже по тексту, а сейчас — вводим код программы:

**format PE GUI 4.0  
entry start  
  
include 'win32a.inc'  
  
section '.data' data readable writeable  
  
class db 'FASMWIN32',0  
title db 'ОКНО',0  
classb db 'BUTTON',0  
classe db 'EDIT',0  
classs db 'STATIC',0  
textb1 db 'Копировать',0  
textb2 db 'Очистить',0  
textg db 'Рамка',0  
texts db 'Текст',0  
textc db 'Очистить все',0  
errtxt db 'Ошибка',0  
hwnd dd ?  
hwnde dd ?  
hwnds dd ?  
hwndc dd ?  
text rb 100  
  
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\_SYSMENU,128,128,256,192,0,0,[wc.hInstance],0  
cmp eax,0  
je error  
mov [hwnd],eax  
msg\_loop:  
invoke GetMessage,msg,0,0,0  
cmp eax,0  
je end\_loop  
invoke IsDialogMessage,[hwnd],msg  
cmp eax,0  
jne msg\_loop  
invoke TranslateMessage,msg  
invoke DispatchMessage,msg  
jmp msg\_loop  
  
error:  
invoke MessageBox,0,errtxt,0,MB\_ICONERROR+MB\_OK  
  
end\_loop:  
invoke ExitProcess,[msg.wParam]  
  
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,textg,WS\_VISIBLE+ WS\_CHILD+ BS\_GROUPBOX,5,5,240,150,[hwnd],1000,[wc.hInstance],0**

**invoke CreateWindowEx,0,classb,textb1,WS\_VISIBLE+WS\_CHILD+ BS\_PUSHBUTTON+WS\_TABSTOP,20,100,100,40,[hwnd],1001,[wc.hInstance],0**

**invoke CreateWindowEx,0,classb,textb2,WS\_VISIBLE+WS\_CHILD+ BS\_PUSHBUTTON+WS\_TABSTOP,130,100,100,40,[hwnd],1002,[wc.hInstance],0**

**invoke CreateWindowEx,0,classe,texts,WS\_VISIBLE+WS\_CHILD+WS\_BORDER+ WS\_TABSTOP+ES\_AUTOHSCROLL,10,25,230,20,[hwnd],1003,[wc.hInstance],0**

**mov [hwnde],eax**

**invoke CreateWindowEx,0,classe,0,WS\_VISIBLE+WS\_CHILD+WS\_BORDER+WS\_TABSTOP+ ES\_AUTOHSCROLL+ES\_READONLY,10,50,230,20,[hwnd],1004,[wc.hInstance],0**

**mov [hwnds],eax**

**invoke CreateWindowEx,0,classb,textc,WS\_VISIBLE+WS\_CHILD+ BS\_AUTOCHECKBOX+WS\_TABSTOP,130,75,110,20,[hwnd],1005,[wc.hInstance],0**

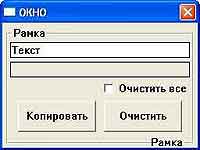
**mov [hwndc],eax**

**invoke CreateWindowEx,0,classs,textg,WS\_VISIBLE+ WS\_CHILD,190,145,43,15,[hwnd],1006,[wc.hInstance],0**

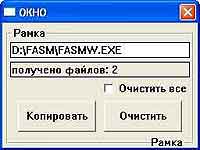
**invoke SetFocus,[hwnde]**

**jmp .finish**

**.wmcommand:  
cmp [wparam],1001  
je .but1  
cmp [wparam],1002  
je .but2  
jmp .finish  
.but1:  
invoke SendMessage,[hwnde],WM\_GETTEXT,100,text  
invoke SendMessage,[hwnds],WM\_SETTEXT,0,text  
jmp .finish  
.but2:  
invoke SendMessage,[hwnds],WM\_SETTEXT,0,0  
invoke SendMessage,[hwndc],BM\_GETCHECK,0,0  
cmp eax,BST\_CHECKED  
jne .finish  
invoke SendMessage,[hwnde],WM\_SETTEXT,0,0  
invoke SendMessage,[hwndc],BM\_SETCHECK,BST\_UNCHECKED,0  
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'**



Если вы все верно набрали, то при запуске у вас появится окошко, как на рис. 1. Понимаю, что набирать большие тексты вручную и без ошибок достаточно трудно, а посему, как обычно, выкладываю копию исходного текста в интернет. Ссылка в конце статьи. Теперь же будем разбирать новый материал. В секции данных внимательный читатель предыдущих статей не найдет ничего нового, но на всякий случай поясню. Инструкции, содержащие директиву db (данные в байтах), объявляют текстовые строки с завершающим ноликом в конце. Инструкции с dd — двойные слова (двойное слово = 4 байта = 32 бита), которые мы резервируем в данном случае для хранения дескрипторов некоторых окон (hwnd — так обычно сокращают handle of window). Инструкция text rb 100 резервирует 100 байт под переменную для хранения текста. Будет вам мало 100 байт — выделите больше. Ну и две структуры данных: WNDCLASS и MSG, о которых подробно говорили в прошлый раз. Окно создается практически как в программе из прошлой статьи. Только теперь нам необходимо сохранить его дескриптор в переменную для последующего использования: mov [hwnd],eax. В цикл обработки сообщений теперь добавлен вызов функции IsDialogMessage. Эта функция определяет, адресовано ли сообщение указанному блоку диалога, и если да — обрабатывает сообщение. Ее параметры: идентификатор окна или блока диалога и указатель на структуру данных MSG. Если сообщение не было обработано, то возвращаемое значение — ноль. Иначе — сообщение было обработано функцией, и нам не надо его повторно обрабатывать. Поэтому, если после выполнения функции eax не равен нолю, цикл начинается сначала, а если ноль, то обработка сообщения продолжается. Использование этой функции требуется, например, для правильной обработки переключения между элементами, имеющими стиль WS\_TABSTOP, по клавише Tab. Уберите ее из цикла (вместе с cmp eax,0 и jne msg\_loop), и переключение не будет осуществляться.



В прошлый раз после метки .wmcreate у нас создавалось лишь одно дочернее окно — кнопка. Теперь у нас их целых семь: рамка (GroupBox), две кнопки (PushButton), два поля редактирования (EditBox), флажок (CheckBox), надпись (Static). Надпись, кстати, имеет такой же текст, что и рамка, и поставил я ее в нижней части рамки. Не перепутайте их. Чтобы определиться, что есть где, можете убрать из параметра стиля WS\_VISIBLE вместе с плюсиком и проверить, что исчезло из окна. **WS\_VISIBLE** указывает, что окно создается видимым для пользователя. WS\_CHILD обозначает дочернее окно. WS\_BORDER создает окно в рамочке. WS\_TABSTOP разрешает переход на данное окно клавишей Tab. В общем, приставка WS\_ — это стиль окна (Window Style), BS\_ — стиль кнопки (окно класса BUTTON), ES\_ — Edit Style и т.д. Причем наше первое дочернее окно — рамка — тоже является окном класса BUTTON со стилем BS\_GROUPBOX. Ее основное предназначение — группировка кнопок в логический блок. Заголовок такого окна (при наличии) отображается в левом верхнем углу рамки. ES\_AUTOHSCROLL разрешает горизонтальную прокрутку текста, когда он выходит за пределы окна. ES\_READONLY запрещает редактирование текста. Я уже говорил о нецелесообразности описания всех стилей в одной статье, поэтому назначение других стилей вам придется узнавать самостоятельно или по ходу рассмотрения их в наших занятиях. Функция **CreateWindowEx** в случае успешного завершения возвращает в eax дескриптор созданного окна. Дескрипторы некоторых окон нам понадобятся для последующих операций с ними, поэтому мы сохраняем дескрипторы этих окон в переменные **hwnde**, **hwnds** и **hwndс**. Функция **SetFocus** активирует указанное окно для ввода с клавиатуры. Поэтому при запуске программы курсор текстового ввода уже приветливо мигает нам из поля ввода текста. Ну вот мы и добрались до ядра нашей программы. При нажатии кнопки Копировать происходит переход на метку .but1. Функция SendMessage отправляет сообщение указанному окну. Ее параметры — соответственно дескриптор окна-получателя; собственно сообщение; первый параметр сообщения; второй параметр сообщения. **WM\_GETTEXT** посылается окну, чтобы скопировать текст окна в указанный во втором параметре буфер. Для этого сообщения в первом параметре необходимо указать максимальное количество копируемых символов (включая завершающий строку ноль), а во втором — указатель на переменную-буфер, в которую будет помещена строка. WM\_SETTEXT устанавливает новым текстом окна строку из буфера, указанного во втором параметре. Первый параметр в этом сообщении не используется, и потому должен быть нулем. Таким образом, на метке .but1 мы копируем до сотни символов из окна hwnde в переменную text, а затем устанавливаем этот текст в качестве нового текста окна hwnds. Причем первая команда наглядно показывает простой метод ввода данных в оперативную память компьютера, а вторая — вывода данных из оперативной памяти в окно программы. По нажатию кнопки Очистить мы переходим на метку .but2. Первой командой мы устанавливаем "пустой" текст в окне hwnds. Далее мы используем сообщение BM\_GETCHECK, предназначенное для проверки состояния флажков (check box) и радиокнопок (radio button). Возвращаемые значения могут быть следующими: BST\_CHECKED, BST\_UNCHECKED и BST\_INDETERMINATE — отмечено, не отмечено и не определено. Параметры не используются. Если состояние не BST\_CHECKED — переход на метку .finish, иначе — выполняем следующие команды: устанавливаем "пустой" текст в окне hwnde и снимаем флажок — сообщение BM\_SETCHECK — выставить состояние флажка или радиокнопки в соответствии с первым параметром. Второй параметр не используется.

Очень удобно ввод данных в программу из какого-либо файла производить перетаскиванием пиктограммы файла на окно программы (Drag and Drop). Для этого прежде всего необходимо разрешить окну получать перетаскиваемые файлы. Это можно сделать, установив соответствующий расширенный стиль окна:

**invoke CreateWindowEx,WS\_EX\_ACCEPTFILES,class,title, WS\_VISIBLE+WS\_SYSMENU,128,128,256,192,0,0,[wc.hInstance],0**  
  
В процедуру обработки сообщений добавим обработку сообщения о получении файлов:  
  
**… … …  
cmp [wmsg],WM\_DROPFILES  
je .drop**  
**… … …**

**.drop:  
invoke DragQueryFile,[wparam],-1,text,100  
invoke wsprintf,text,form,eax  
invoke SendMessage,[hwnds],WM\_SETTEXT,0,text  
invoke DragQueryFile,[wparam],0,text,100  
invoke SendMessage,[hwnde],WM\_SETTEXT,0,text  
jmp .finish  
… … …**  
Функция DragQueryFile содержится в библиотеке shell32, поэтому в секцию импортируемых данных добавляем импорт shell32:  
section '.idata' import data readable writeable  
  
**library kernel32,'KERNEL32.DLL',\  
shell32,'SHELL32.DLL',\  
user32,'USER32.DLL'  
  
include 'api\kernel32.inc'  
include 'api\shell32.inc'  
include 'api\user32.inc'**  
  
Ну и для вывода сообщения о количестве полученных файлов добавим образец формата в секцию данных:  
  
**form db 'получено файлов: %d',0**

Теперь при перетаскивании файла или нескольких файлов в область главного окна окно получает сообщение WM\_DROPFILES, и программа переходит на метку .drop. Функция **DragQueryFile** сообщает в указанный буфер имя полученного файла. Ее параметры следующие: идентификатор структуры, содержащей полученные файлы, сообщенный программе в первом параметре WM\_DROPFILES; порядковый номер полученного файла; буфер для ответа; размер буфера. Следует обратить внимание, что, если указать в качестве порядкового номера -1, то в регистр eax будет возвращено количество полученных файлов. Отсчет порядковых номеров начинается с нуля, то есть ноль — это первый полученный файл. Если по верно указанному порядковому номеру в буфер возвращается ноль — значит, в буфере недостаточно места для размещения имени файла, и в eax возвращается необходимый размер в символах без учета завершающего строку нуля. Иначе в eax возвращается число скопированных символов без учета завершающего нуля. Функция **wsprintf** форматирует блоки данных в соответствии с указанным форматом и помещает отформатированную строку в буфер ответа. Параметры: указатель на буфер для ответа; указатель на образец формата; один или несколько блоков данных через запятую, которые будут отформатированы и помещены в буфер ответа. При успешном выполнении в eax возвращается количество символов, помещенных в буфер ответа. Простейший образец формата — строка, содержащая символ процента и символ — тип формата. %d обозначает формат десятичного числа с учетом знака (эквивалент %i), %u — целое число без учета знака, %x и %X — шестнадцатеричное число соответственно в нижнем и верхнем регистре, %s — строка символов (рис. 2). Таким образом, если перетащить файл или файлы на главное окно нашей программы, в окно hwnds будет помещена строка, сообщающая количество полученных файлов, а в окне hwnde отобразится имя первого из полученных файлов. Теперь вы имеете общее представление о том, как можно открыть тот или иной файл в программе, просто перетащив его на окно этой программы. Многое еще не ясно, но мы обязательно поговорим об этом в следующих статьях.  
  
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 SASecurity gr.**

*Компьютерная газета. Статья была опубликована в номере 20 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/)