UIC – Università Italiana Cracking

http://quequero.org

Moving to Windows Vista x64

written by Ntoskrnl

UIC New Year Pack 2 – 01/Jan/2007

Moving to Windows x64

Index

Introduction

This is an introduction to Windows Vista and the x64 architecture. Writing an article like this is always uneasy, because there's plenty to talk about, but on the other hand it's an article, not a book. I tried to focus on some important aspects, but it goes without saying it that I had to cut out a lot (e.g. the User-Mode Driver Framework, and I'm very sorry for that). This is just a general overview on certain topics, if you want to learn more, then you should really consider turning to specific guides. Also, I won't talk about some obvious matters of the x64 architecture, like the fact that applications can now access a larger memory range etc. This article should be considered a quick upgrade for x86/XP developers.

At the time I write this article I've been using Windows Vista for a month and its official release is scheduled for January 30th (so, in another month). I moved to x64 with XP some months ago and at the time I did I was surprised that I found all the drivers for my devices. But, as we know, Windows Vista requires drivers to be certified, and in order to get the certification companies have to supply a x64 version of the driver. No certification will be released for x86-only drivers. However, at the moment I write, a lot of applications like virtual drive encrypters don't provide drivers for Vista (since x64 versions haven't got a certificate). If you didn't know about the certification, don't worry, I'll talk about it later and you'll see that it's still possible to run drivers without it. I just wanted to say that hardware compatibility is no longer an issue like it was one year ago, and by switching to Windows Vista x64 you're not taking too much chances.

I tried to organize this article in two sections, one about the changes brought us by x64 and then by Vista. I tried as hard as possible to separate these two things, because the x64 technology already existed under Windows XP, so it was important to me that the reader was given a clear distinction

between those things that affect only Vista and those ones which affect both topics.

x64 Section

x64 Assembly

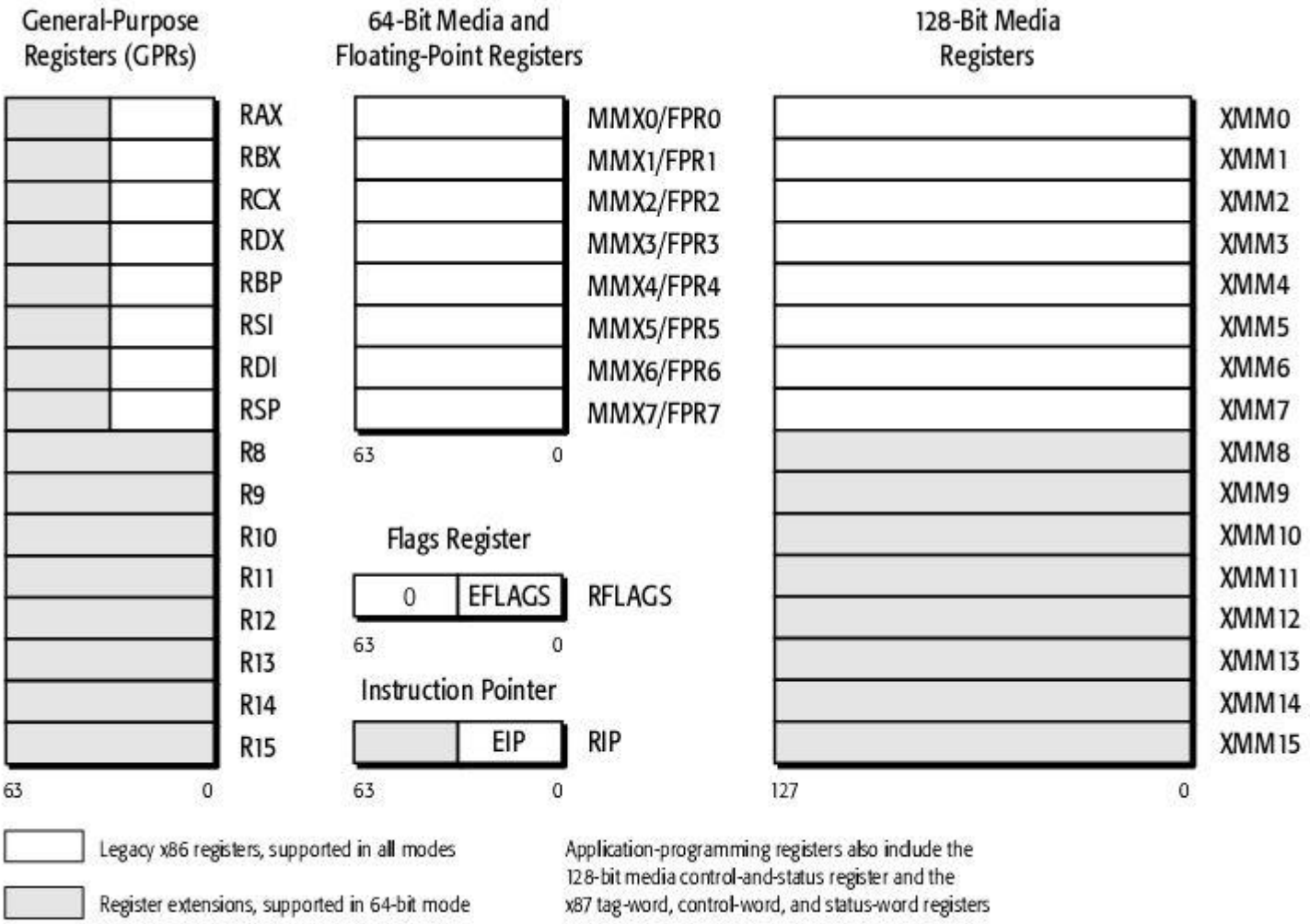
In this paragraph I'll try to explain the basics of x64 assembly. I assume the reader is already familiar with x86 assembly, otherwise he won't be able to make heads or tails of this paragraph. Moreover, since this is just a very (but very) brief guide, you'll have to look into the AMD64 documentation for more advanced stuff. Some stuff I won't even mention, you'll see by yourself that some instructions are no longer in use: for instance, that the lea instruction has completely taken place of the mov offset.

What you're going to notice at once is that there are some more registers in the x64 syntax:

- 8 new general-purpose registers (GPRs).
- 8 new 128-bit XMM registers.

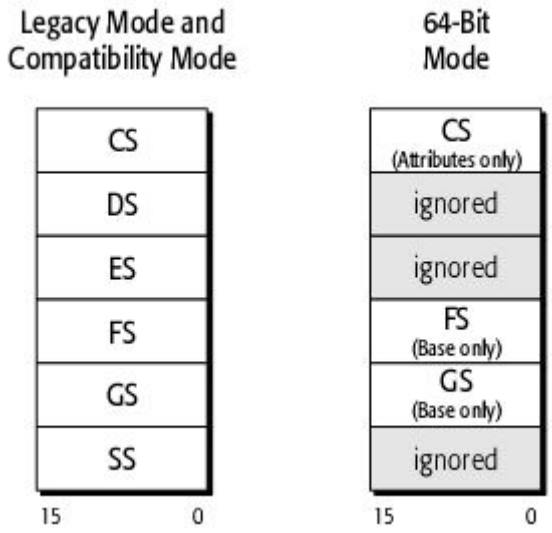
Of course, all general-purpose registers are 64 bits wide. The old ones we already knew are easy to recognize in their 64-bit form: rax, rbx, rcx, rdx, rsi, rdi, rbp, rsp (and rip if we want to count the instruction pointer). These old registers can still be accessed in their smaller bit ranges, for instance: rax, eax, ax, ah, al. The new registers go from r8 to r15, and can be accessed in their various bit ranges like this: r8 (qword), r8d (dword), r8w (word), r8b (low byte).

Here's a figure taken from the AMD docs:

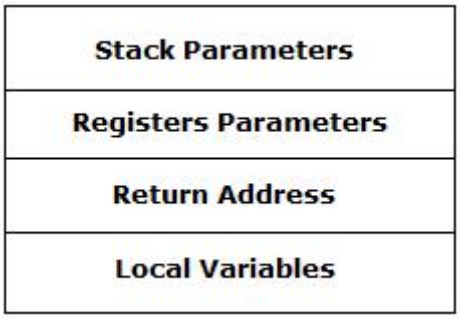


Applications can still use segments registers as base for addressing, but the 64-bit mode only

recognizes three of the old ones (and only two can be used for base address calculations). Here's another figure:



And now, the most important things. Calling convention and stack. x64 assembly uses FASTCALLs as calling convention, meaning it uses registers to pass the first 4 parameters (and then the stack). Thus, the stack frame is made of: the stack parameters, the registers parameters, the return address (which I remind you is a qword) and the local variables. The first parameter is the rcx register, the second one rdx, the third r8 and the fourth r9. Saying that the parameters registers are part of the stack frame, makes it also clear that any function that calls another child function has to initialize the stack providing space for these four registers, even if the parameters passed to the child function are less than four. The initialization of the stack pointer is done only in the prologue of a function, it has to be large enough to hold all the arguments passed to child functions and it's always a duty of the caller to clean the stack. Now, the most important thing to understand how the space is provided in the stack frame is that the stack has to be 16-byte aligned. In fact, the return address has to be aligned to 16 bytes. So, the stack space will always be something like 16n + 8, where n depends on the number of parameters. Here's a small figure of a stack frame:



Don't worry if you haven't completely figured out how it works: now we will see a few code samples, which, in my opinion, always make the theory a lot easier to understand. Let us take for instance a hello-world application like:

```
int WINAPI _tWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR szCmdLine,
int iCmdShow)
{
    MessageBox(NULL, _T("Hello World!"), _T("My First x64 Application"), 0);
    return 0;
}
```

This code disassembled would look like:

```
.text:0000000000401220 sub_401220 proc near          ; CODE XREF: start+10E p
.text:0000000000401220
.text:0000000000401220 arg_0= qword ptr 8
.text:0000000000401220 arg_8= qword ptr 10h
```

```
.text:0000000000401220 arg_10= qword ptr 18h
.text:0000000000401220 arg_18= dword ptr 20h
.text:0000000000401220
.text:0000000000401220    mov [rsp+arg_18], r9d
.text:0000000000401225    mov [rsp+arg_10], r8
.text:000000000040122A    mov [rsp+arg_8], rdx
.text:000000000040122F    mov [rsp+arg_0], rcx
.text:0000000000401234    sub rsp, 28h
.text:0000000000401238    xor r9d, r9d             ; uType
.text:000000000040123B    lea r8, Caption          ; "My First x64 Application"
.text:0000000000401242    lea rdx, Text            ; "Hello World!"
.text:0000000000401249    xor ecx, ecx             ; hWnd
.text:000000000040124B    call cs:MessageBoxA
.text:0000000000401251    xor eax, eax
.text:0000000000401253    add rsp, 28h
.text:0000000000401257    retn
.text:0000000000401257 sub_401220 endp
```

The stack pointer initialization is all about the things I said earlier. Since we are calling a child-function with parameters we need the space for all four parameter registers (0x20, this value is already aligned to 16 byte) and the return address (0x08). Thus, we'll have 0x28. Remember that if the stack-value is too small or is not aligned, your code will crash at once. Also, don't wonder why there's no ExitProcess in this function: compiling the code above with Visual C++ adds always a stub (WinMainCRTStartup) which then calls our WinMain. So, the ExitProcess is in the stub code. But what happens when the code before the MessageBox calls a function which take seven parameters instead of four?

```
.text:0000000000401180 sub_401180 proc near         ; CODE XREF: sub_4011F0+4 p
.text:0000000000401180                              ; sub_4011F0+11 p
.text:0000000000401180
.text:0000000000401180 var_28= qword ptr -28h
.text:0000000000401180 var_20= qword ptr -20h
.text:0000000000401180 var_18= qword ptr -18h
.text:0000000000401180
.text:0000000000401180    sub rsp, 48h
.text:0000000000401184    lea rax, unk_402040
.text:000000000040118B    mov [rsp+48h+var_18], rax
.text:0000000000401190    lea rax, unk_402044
.text:0000000000401197    mov [rsp+48h+var_20], rax
.text:000000000040119C    lea rax, unk_402048
.text:00000000004011A3    mov [rsp+48h+var_28], rax
.text:00000000004011A8    lea r9, qword_40204C       ; __int64
.text:00000000004011AF    lea r8, qword_40204C+4     ; __int64
.text:00000000004011B6    lea rdx, unk_402054        ; __int64
.text:00000000004011BD    lea rcx, aAa               ; "ptr"
.text:00000000004011C4    call TakeSevenParameters
.text:00000000004011C9    xor r9d, r9d               ; uType
.text:00000000004011CC    lea r8, Caption            ; "My First x64 Application"
.text:00000000004011D3    lea rdx, Text              ; "Hello World!"
.text:00000000004011DA    xor ecx, ecx               ; hWnd
.text:00000000004011DC    call cs:MessageBoxA
.text:00000000004011E2    add rsp, 48h
.text:00000000004011E6    retn
.text:00000000004011E6 sub_401180 endp
```

As said, the child function takes 7 parameters, making it necessary to provide space for 3 extra parameters on the stack. So, 7 * 8 = 0x38, which aligned to 16byte is 0x40. Providing, then, space for the return address makes it 0x48, our value indeed. I think you have understood the stack-frames logic by now, it's actually quite easy to understand it, but it needs a second to revert from the old x86/stdcall logic to this one. But now enough of this, now that we've seen how the x64 code works, we'll try compiling an assembly source by ourselves.

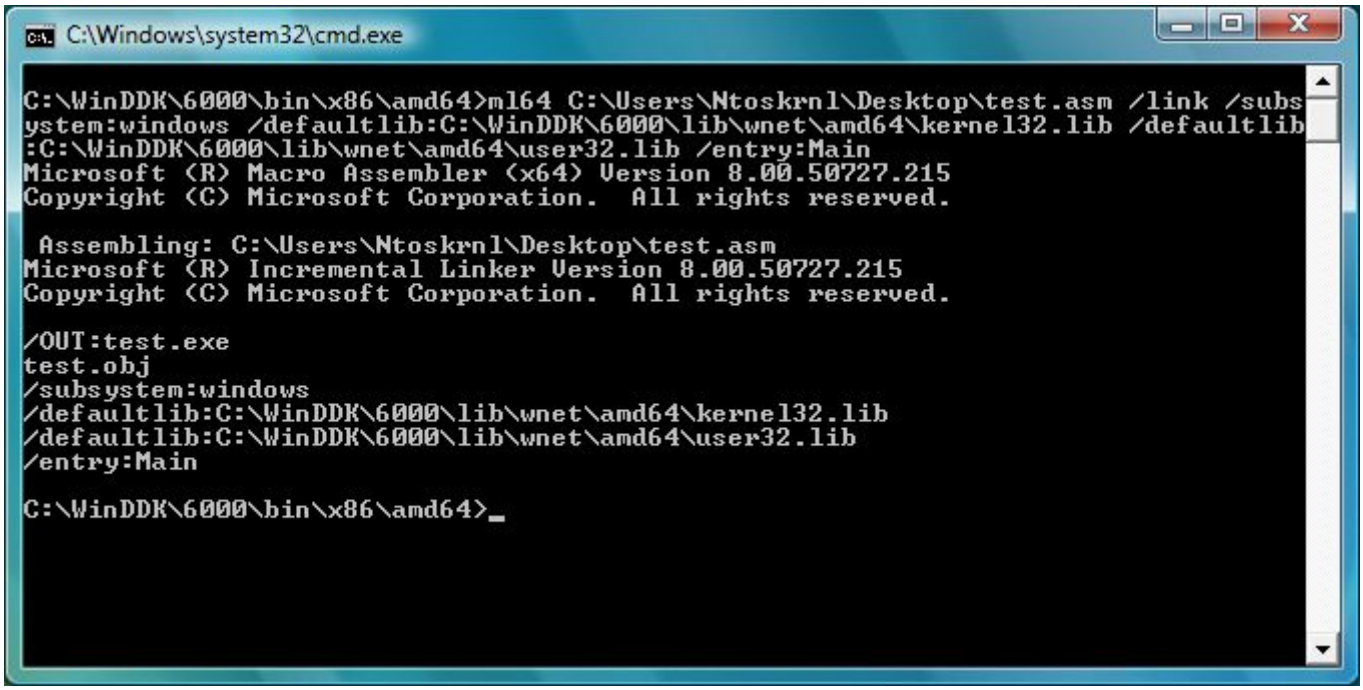
Before we start, I have to make something clear. There are some assemblers over the internet which make the job easier, mainly because the initialize the stack by themselves or they create code that is easy to converto from/to x86. But I think that is not the point here in this article. In fact, I'm going to use the microsoft assembler (ml64.exe), which requires you to write everything down, just like in the disassembly. Another option could be compiling the with another assembler and then link it with ml64. I think the reader should really make these decisions on his own. As far as I am

concerned, I don't believe that much code should be written in assembly and avoided whenever it could be done. This new x64 technology is a good opportunity to re-think about these matters. In the last years I always wrote 64-bit compatible code in C/C++ (I mean unmanaged, of course) and when I had to recompile a project of 70,000 lines of code for x64, I didn't had to change one single line of code (I'll talk about the C/C++ programming later). Despite of all the macros an assembler offers, I seriously doubt that people who wrote their whole code in assembly will be able to switch so easily to x64 (remember one day even the IA64 syntax could be adopted). I think in most cases the obvious choice will be not converting to the new technology and stick to x86, but this isn't always possible, it depends on the software category.

The microsoft assembler is contained in the SDK and in the DDK (WDK for Vista). Right now, I'm using Vista's WDK, which I freely downloaded from the msdn. The first sample of code I'm going to show you is a simple Hello-World messagebox application.

```
extrn MessageBoxA : proc
extrn ExitProcess : proc

.data
body db 'Hello World!', 0
capt db 'My First x64 Application', 0

.code
Main proc
sub rsp, 28h
xor r9d, r9d        ; uType = 0
lea r8, capt        ; lpCaption
lea rdx, body       ; lpText
xor rcx, rcx        ; hWnd = NULL
call MessageBoxA
xor ecx, ecx        ; exit code = 0
call ExitProcess
Main endp

end
```

As you can see, I didn't bother unwinding the stack, since I call ExitProcess. The syntax is very similar to the old MASM one, although there are a few dissimalirites. The ml64 console output should be something like this:



The command line to compile is:

ml64 C:\...\test.asm /link /subsystem:windows

/defaultlib:C:\WinDDK\6000\lib\wnet\amd64\kernel32.lib
/defaultlib:C:\WinDDK\6000\lib\wnet\amd64\user32.lib /entry:Main

If the libs are not in the same directory as ml64.exe, you'll have to provide the path like I did. The entry has to be provided, otherwise you would have to use WinMainCRTStartup as main entry.

The next sample of code I'm going to show you displays a window calling CreateWindowEx. What you're going to learn through this code is structure alignment and how integrating resources in your projects. Like I said earlier, I don't want to encourage you to write your windows in assembly, but I believe that this sort of code is good for learning. Now the code, afterwards the explanation.

Open Test.zip (16kb) from “Files” directory inside the package.

test.zip ----------

```
extrn GetModuleHandleA : proc
extrn MessageBoxA : proc
extrn RegisterClassExA : proc
extrn CreateWindowExA : proc
extrn DefWindowProcA : proc
extrn ShowWindow : proc
extrn GetMessageA : proc
extrn TranslateMessage : proc
extrn DispatchMessageA : proc
extrn PostQuitMessage : proc
extrn DestroyWindow : proc
extrn ExitProcess : proc

WNDCLASSEX struct
  cbSize            dd      ?
  style             dd      ?
  lpfnWndProc       dq      ?
  cbClsExtra        dd      ?
  cbWndExtra        dd      ?
  hInstance         dq      ?
  hIcon             dq      ?
  hCursor           dq      ?
  hbrBackground     dq      ?
  lpszMenuName      dq      ?
  lpszClassName     dq      ?
  hIconSm           dq      ?
WNDCLASSEX ends

POINT struct
  x                 dd      ?
  y                 dd      ?
POINT ends

MSG struct
  hwnd              dq      ?
  message           dd      ?
  padding1          dd      ?       ; padding
  wParam            dq      ?
  lParam            dq      ?
  time              dd      ?
  pt                POINT   <>
  padding2          dd      ?       ; padding
MSG ends

.const
NULL equ 0
CS_VREDRAW equ 1
CS_HREDRAW equ 2
COLOR_WINDOW equ 5
; WS_OVERLAPPEDWINDOW = (WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU | WS_THICKFRAME |
WS_MINIMIZEBOX | WS_MAXIMIZEBOX)
WS_OVERLAPPEDWINDOW equ 0CF0000h
```

```
CW_USEDEFAULT equ 80000000h
SW_SHOW equ 5
WM_DESTROY equ 2
WM_COMMAND equ 111h
IDC_MENU equ 109
IDM_ABOUT equ 104
IDM_EXIT equ 105

.data
szWindowClass db 'FirstApp', 0
szTitle db 'My First x64 Windows', 0
szHelpTitle db 'Help', 0
szHelpText db 'This will be a big help...', 0

.data?
hInstance qword ?
hWnd qword ?
wndclass WNDCLASSEX <>
wmsg MSG <>

.code

WndProc: ; proc hWnd : qword, uMsg : dword, wParam : qword, lParam : qword
  mov [rsp+8], rcx        ; hWnd (save parameters as locals)
  mov [rsp+10h], edx      ; Msg
  mov [rsp+18h], r8       ; wParam
  mov [rsp+20h], r9       ; lParam
  sub rsp, 38h
  cmp edx, WM_DESTROY
  jnz @next1

  xor ecx, ecx          ; exit code
  call PostQuitMessage
  xor rax, rax
  ret

@next1:
  cmp edx, WM_COMMAND
  jnz @default

  mov rbx, rsp
  add rbx, 38h
  mov r10, [rbx+18h]     ; wParam
  cmp r10w, IDM_ABOUT
  jz @about
  cmp r10w, IDM_EXIT
  jz @exit
  jmp @default

@about:
  xor r9d, r9d
  lea r8, szHelpTitle
  lea rdx, szHelpText
  xor ecx, ecx
  call MessageBoxA
  jmp @default

@exit:
  mov rbx, rsp
  add rbx, 38h
  mov rcx, [rbx+8h]       ; hWnd
  call DestroyWindow

@default:
  mov rbx, rsp
  add rbx, 38h
  mov r9, [rbx+20h]       ; lParam
  mov r8, [rbx+18h]       ; wParam
  mov edx, [rbx+10h]      ; Msg
```

```
  mov rcx, [rbx+8]        ; hWnd
  call DefWindowProcA
  add rsp, 38h
  ret

MyRegisterClass:  ; proc hInst : qword
  sub rsp, 28h
  mov wndclass.cbSize, sizeof WNDCLASSEX
  mov eax, CS_VREDRAW
  or eax, CS_HREDRAW
  mov wndclass.style, eax
  lea rax, WndProc
  mov wndclass.lpfnWndProc, rax
  mov wndclass.cbClsExtra, 0
  mov wndclass.cbWndExtra, 0
  mov wndclass.hInstance, rcx
  mov wndclass.hIcon, NULL
  mov wndclass.hCursor, NULL
  mov wndclass.hbrBackground, COLOR_WINDOW
  mov wndclass.lpszMenuName, IDC_MENU
  lea rax, szWindowClass
  mov wndclass.lpszClassName, rax
  mov wndclass.hIconSm, NULL
  lea rcx, wndclass
  call RegisterClassExA
  add rsp, 28h
  ret


InitInstance: ; proc hInst : qword
  sub rsp, 78h
  mov rax, CW_USEDEFAULT
  xor rbx, rbx
  mov [rsp+58h], rbx            ; lpParam
  mov [rsp+50h], rcx            ; hInstance
  mov [rsp+48h], rbx            ; hMenu = NULL
  mov [rsp+40h], rbx            ; hWndParent = NULL
  mov [rsp+38h], rbx            ; Height
  mov [rsp+30h], rax            ; Width
  mov [rsp+28h], rbx            ; Y
  mov [rsp+20h], rax            ; X
  mov r9d, WS_OVERLAPPEDWINDOW  ; dwStyle
  lea r8, szTitle               ; lpWindowName
  lea rdx, szWindowClass        ; lpClassName
  xor ecx, ecx                  ; dwExStyle
  call CreateWindowExA
  mov hWnd, rax
  mov edx, SW_SHOW
  mov rcx, hWnd
  call ShowWindow
  mov rax, hWnd                 ; set return value
  add rsp,78h
  ret


Main proc
  sub rsp, 28h
  xor rcx, rcx
  call GetModuleHandleA
  mov hInstance, rax
  mov rcx, rax
  call MyRegisterClass
  test rax, rax
  jz @close             ; if the RegisterClassEx fails, exit

  mov rcx, hInstance
  call InitInstance
  test rax, rax
  jz @close             ; if the InitInstance fails, exit
```

```
@handlemsgs:            ; message processing routine
  xor r9d, r9d
  xor r8d, r8d
  xor edx, edx
  lea rcx, wmsg
  call GetMessageA
  test eax, eax
  jz @close
  lea rcx, wmsg
  call TranslateMessage
  lea rcx, wmsg
  call DispatchMessageA
  jmp @handlemsgs

@close:
  xor ecx, ecx
  call ExitProcess
Main endp

end
```

As you can see, I tried to stay as low level as I could. The reason why I avoided for other functions other than the main the proc macro is that the ml64 puts a prologue end an epilogue, which I didn't want, by itself. Avoiding the macro made it possible to define my own stack frame without any intermission by the compiler. The first thing to notice scrolling this code is the structure:

```
MSG struct
  hwnd              dq      ?
  message           dd      ?
  padding1          dd      ?       ; padding
  wParam            dq      ?
  lParam            dq      ?
  time              dd      ?
  pt                POINT   <>
  padding2          dd      ?       ; padding
MSG ends
```

It requires two paddings which the x86 declaration of the same structure didn't. The reason, in a few words, is that qword members should be aligned to qword boundaries (this for the first padding). The additional padding at the end of the structure follows the rule that: every structure should be aligned to its largest member. So, being its largest member a qword, the structure should be aligned to an 8-byte boundary.

To compile this sample, the command line is:

ml64 c:\myapp\test.asm /link /subsystem:windows
/defaultlib:C:\WinDDK\6000\lib\wnet\amd64\kernel32.lib
/defaultlib:C:\WinDDK\6000\lib\wnet\amd64\user32.lib /entry:Main c:\myapp\test.res

test.res is a file I took from a VC++ wizard project, I was too lazy to make on by myself. Anyway, making a resource file is very easy with the VC++, but no one forbids you to use the notepad, it just takes more time. To compile the resource file all you need to do is to use the command line: "rc test.rc".

I think the rest of the code is pretty easy to understand. I didn't cover everything with this paragraph, but now you should have quite a good insight into x64 assembly. Let's move on.

C/C++ Programming

Writing x64 compatible code in C/C++ is very easy. All what it takes is to follow some basic rules.

The most common mistake that make that makes 99% of the old 32bit sources uncompatible is wrong casting. For Instance:

```
ptr1 = (DWORD) (sizoef (x) + ptr2);  <-- WRONG!
```

This line of code assumes that pointers are 32bit long, but on x64 pointers are 64bit long and the line of code above basically truncates the pointer making it invalid. So, always cast like this:

```
ptr1 = (ULONG_PTR) (sizoef (x) + ptr2);  <-- RIGHT!
```

It doesn't matter if you use ULONG_PTR, LONG_PTR, DWORD_PTR or whatever. The important thing is that you use one of these defines (or directly by pointer type: (void *)).

Keep in mind that all handles and handle derivates are qwords. HANDLE, HKEY, HICON, HBITMAP, HINSTANCE, HMODULE, HWND etc. etc. These are all 64bit long, even though they're not all the same handle (HINSTANCE, for example, is just a pointer, not a real handle). Even WPARAM and LPARAM are now 64bit long. There's no rule to follow, just don't assume these types are 32 or 64bit long: write code that is compatible with both conditions:

```
HWND *hWndArray = (HWND *) malloc(sizeof (DWORD) * n);  <-- WRONG!
```

Instead write:

```
HWND *hWndArray = (HWND *) malloc(sizeof (HWND) * n);  <-- RIGHT!
```

As you can see this isn't a rule, just good sense.

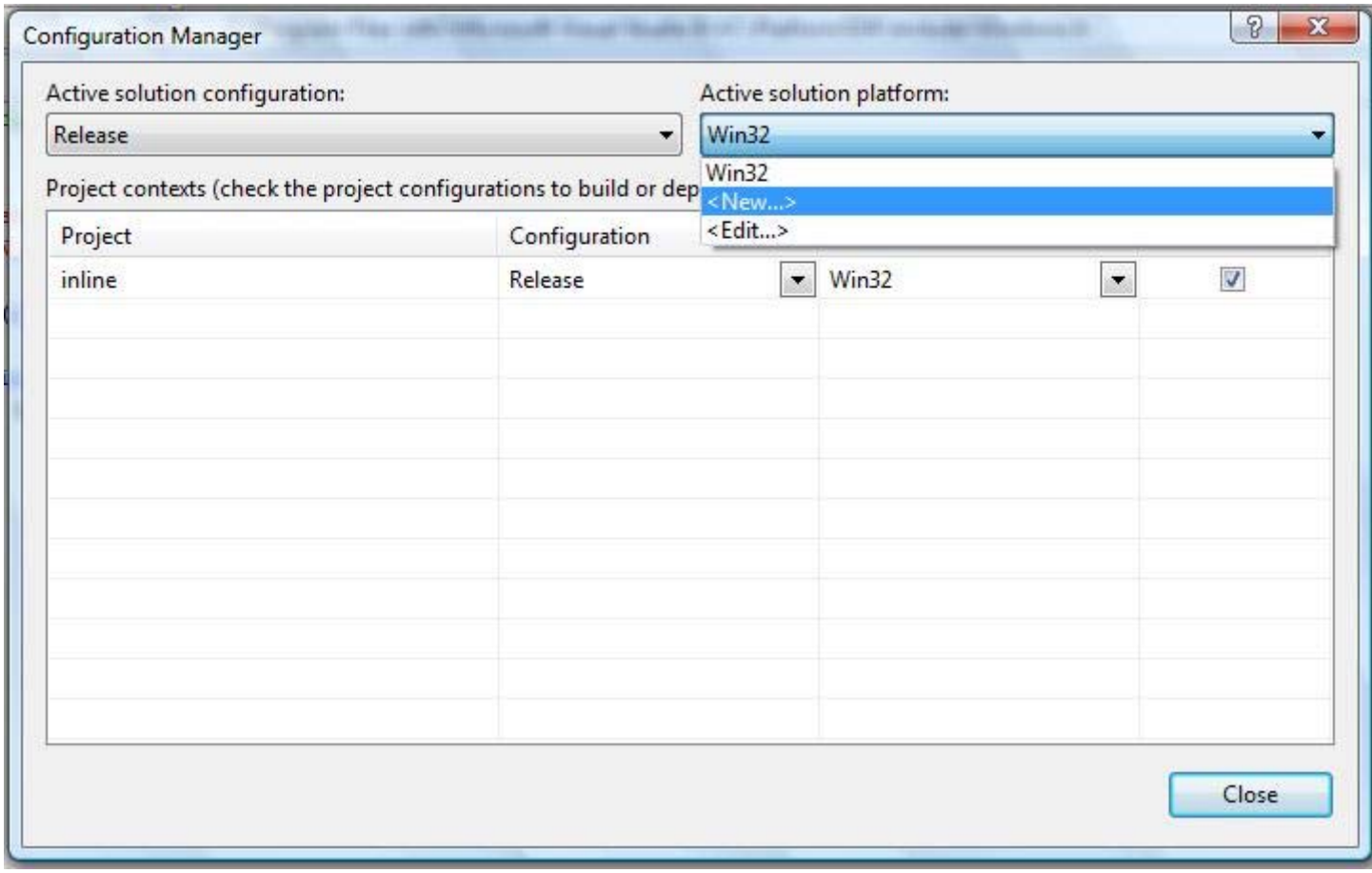
The defines to use for writing architecture-dependent code are:

_M_IX86	x86 code only.
_M_AMD64	x64 code only.
_M_IA64	Itanium code only.
_WIN32	32bit code (x86, maybe ARM for WINCE).
_WIN64	64bit code (x64, Itanium).

if you want to write, for example, a piece of code for x86 only, you could write:

```
#ifdef _M_IX86
    // x86 only code
#endif
```

Now that you know all the rules, you just have to compile your project for x64. Keep in mind that every project in VC++ (nowadays) starts with a x86 configuration: it's your job to add a project configuration to the project, but don't worry it's very easy. All you have to do is open the configuration manager (Build -> Configuration Manager) and then under "Active solution platform" click New, just like this:



A dialog box will pop up where you can choose the new platform which for to create a new project configuration. There's nothing more to do, except to build.

Inline Assembly

Bad news! Microsoft completely removed the support for inline assembly in C/C++, both for user and kernel mode. If you try to compile a code sample like this on x64/Itanium:

```
int _tmain(int argc, _TCHAR* argv[])
{
    __asm int 3;
    return 0;
}
```

It will give you more than just one error. Being the __asm keyword no longer supported, the __naked declspec was removed as well (since it doesn't make sense without inline assembly).

Now, prepare for the good news. Before you start thinking about using external asm files or stuff like that, you should know that the VC++ offers some very powerful assembly intrinsics. The header to include to use these intrinsics is "intrin.h". Let's take for a code sample the intrinsics _ReturnAddress() and _AddressOfReturnAddress(). The first one gives us the return address of the current function and the second one the address of the return address itself. Let's analyze this little code sample that I took from the MSDN:

```
int _tmain(int argc, _TCHAR* argv[])
{
    void* pvAddressOfReturnAddress = _AddressOfReturnAddress();

  printf_s("%p\n", pvAddressOfReturnAddress);
    printf_s("%p\n", *((void**) pvAddressOfReturnAddress));
    printf_s("%p\n", _ReturnAddress());

    return 0;
```

```
}
```

The second and the third printf_s will show the same output, since both display the return address of the current function. These intrinsics are very powerful, and nothing can stop us from doing some of the old tricks we did with inline assembly. For instance, having the address of the return address could give me the possibility of changing it and making the function return somewhere else. Let's try that:

```
ULONG_PTR OldAddress = 0;

void f1()
{
    printf_s("Hello there!\n");

    ULONG_PTR *pAddressOfReturnAddress = (ULONG_PTR *) _AddressOfReturnAddress();

    if (OldAddress == 0)
    {
        OldAddress = *pAddressOfReturnAddress;
        *pAddressOfReturnAddress = (ULONG_PTR) &f1;
    }
    else
    {
        *pAddressOfReturnAddress = OldAddress;
    }
}
```

The output of this function is:

```
Hello there!
Hello there!
```

That's because, as you can see from the code, I changed the return address of the current function making it execute again. I put a condition to make it execute again just once, otherwise it would have brought to an endless loop. An important thing to know is that this sample works in Release mode only if you disable code optimization, otherwise the VC++ will remove the line of code which sets the new return address. I'm sure there are ways to trick the VC++ not to do this, but the problem is that if the function is called just by one caller like this one, the VC++ will put the code of the function directly in the caller one, so setting a new return address under these conditions is a bit risky. Disabling optimization is, I believe, the safest way to act.

Enough of this trivia. Here's a list of the intrinsics for x64 taken from the MSDN (many of them are supported on x86 as well):

_AddressOfReturnAddress	Provides the address of the memory location that holds the return address of the current function. This address may not be used to access other memory locations (for example, the function's arguments).
__addgsbyte, __addgsword, __addgsdword, __addgsqword	Add a value to a memory location specified by an offset relative to the beginning of the **GS** segment.
__assume	Passes a hint to the optimizer.
_BitScanForward, _BitScanForward64	Search the mask data from least significant bit (LSB) to the most significant bit (MSB) for a set bit (1).

_BitScanReverse, _BitScanReverse64	Search the mask data from most significant bit (MSB) to least significant bit (LSB) for a set bit (1).
_bittest, _bittest64	Generates the **bt** instruction, which examines the bit in position b of address a, and returns the value of that bit.
_bittestandcomplement, _bittestandcomplement64	Generate the **btc** instruction, which examines bit b of the address a, returns its current value, and sets the bit to its complement.
_bittestandreset, _bittestandreset64	Generate the **btr** instruction, which examines bit b of the address a, returns its current value, and resets the bit to 0.
_bittestandset, _bittestandset64	Generate the **bts** instruction, which examines bit b of the address a, returns its current value, and sets the bit to 1.
__debugbreak	Causes a breakpoint in your code, where the user will be prompted to run the debugger.
_disable	Disables interrupts.
__emul, __emulu	Performs multiplications that overflow what a 32-bit integer can hold.
_enable	Enables interrupts.
__faststorefence	Guarantees that every preceding store is globally visible before any subsequent store.
__getcallerseflags	Returns the EFLAGS value from the caller's context.
__inbyte	Generates the **in** instruction, returning one byte read from the port specified by Port.
__inbytestring	Reads data from the specified port using the **rep insb** instruction.
__incgsbyte, __incgsword, __incgsdword, __incgsqword	Add one to the value at a memory location specified by an offset relative to the beginning of the **GS** segment.
__indword	Reads one double word of data from the specified port using the **in** instruction.
__indwordstring	Reads data from the specified port using the **rep insd** instruction.
__int2c	Generates the **int 2c** instruction, which triggers the **2c** interrupt.
_InterlockedAnd, _InterlockedAnd64	Used to perform an atomic AND operation on a variable shared by multiple threads.
_interlockedbittestandreset,	Generate the **lock_btr** instruction, which examines bit b of the address

_interlockedbittestandreset64	a and returns its current value.
_interlockedbittestandset, _interlockedbittestandset64	Generate the **lock_bts** instruction, which examines bit b of the address a and returns its current value.
_InterlockedCompareExchange, _InterlockedCompareExchange64, _InterlockedCompare64Exchange128, _InterlockedCompare64Exchange128_acq, _InterlockedCompare64Exchange128_rel	Provides compiler intrinsic support for the Win32 Platform SDK InterlockedCompareExchange function.
_InterlockedCompareExchangePointer	Perform an atomic exchange operation, which copies the address passed in as the second argument to the first and returns the original address of the first.
_InterlockedDecrement, _InterlockedDecrement64	Provides compiler intrinsic support for the Win32 Platform SDK InterlockedDecrement function.
_InterlockedExchange, _InterlockedExchange64	Provide compiler intrinsic support for the Win32 Platform SDK InterlockedExchange function.
_InterlockedExchangeAdd, _InterlockedExchangeAdd64	Provide compiler intrinsic support for the Win32 Platform SDK _InterlockedExchangeAdd Intrinsic Functions function.
_InterlockedExchangePointer	Perform an atomic exchange operation, which copies the address passed in as the second argument to the first and returns the original address of the first.
_InterlockedIncrement, _InterlockedIncrement64	Provide compiler intrinsic support for the Win32 Platform SDK InterlockedIncrement function.
_InterlockedOr, _InterlockedOr64	Perform an atomic operation (in this case, the **OR** operation) on a variable shared by multiple threads.
_InterlockedXor, _InterlockedXor64	Used to perform an atomic operation (in this case, the exclusive or **XOR** operation) on a variable shared by multiple threads.
__invlpg	Generates the x86 **invlpg** instruction, which invalidates the translation lookaside buffer (TLB) for the page associated with memory pointed to by Address.
__inword	Reads data from the specified port using the **in** instruction.
__inwordstring	Reads data from the specified port using the **rep insw** instruction.

__ll_lshift	Shifts a 64-bit value specified by the first parameter to the left by a number of bits specified by the second parameter.
__ll_rshift	Shifts a 64-bit value specified by the first parameter to the right by a number of bits specified by the second parameter.
__load128, __load128_acq	Loads a 128-bit value atomically.
_mm_cvtsd_si64x	Generates the x64 extended form of the Convert Scalar Double-Precision Floating-Point Value to 64-Bit Integer (**cvtsd2si**) instruction, which takes the double in the first element of value and converts it to a 64-bit integer.
_mm_cvtsi128_si64x	Generates the x64 extended form of the **movd** instruction, which extracts the low 64-bit integer from an __**m128i** structure.
_mm_cvtsi64x_sd	Generates the Convert Double Word Integer to Scalar Double-Precision Floating-Point Value (**cvtsi2sd**) instruction.
_mm_cvtsi64x_si128	Generates the x64 extended form of the **movd** instruction, which copies a 64-bit value to a __**m128i** structure, which represents an XMM register.
_mm_cvtsi64x_ss	Generates the x64 extended version of the Convert 64-Bit Integer to Scalar Single-Precision Floating-Point Value (**cvtsi2ss**) instruction.
_mm_cvtss_si64x	Generates the x64 extended version of the Convert Scalar Single Precision Floating Point Number to 64-bit Integer (**cvtss2si**) instruction.
_mm_cvttsd_si64x	Generates the x64 extended version of the Convert with Truncation Scalar Double-Precision Floating-Point Value to 64-Bit Integer (**cvttsd2si**) instruction, which takes the first double in the input structure of packed doubles, converts it to a 64-bit integer, and returns the result.
_mm_cvttss_si64x	Emits the x64 extended version of the Convert with Truncation Single-Precision Floating-Point Number to 64-Bit Integer (**cvttss2si**) instruction.
_mm_set_epi64x	Returns the __**m128i** structure with its two 64-bit integer values initialized to the values of the two 64-bit integers passed in.
_mm_set1_epi64x	Provides a way to initialize the two 64-bit elements of the __**m128i** structure with two identical integers.
_mm_setl_epi64	Returns the lower 64 bits of source argument in the lower 64 bits of the result.
_mm_stream_si64x	Writes the data in Source to a memory location specified by Dest, without polluting the caches.

__movsb	Generates a Move String (**rep movsb**) instruction.
__movsd	Generates a Move String (**rep movsd**) instruction.
__movsq	Generates a repeated Move String (**rep movsq**) instruction.
__movsw	Generates a Move String (**rep movsw**) instruction.
__mul128	Multiplies two 64-bit integers passed in as the first two arguments and puts the high 64 bits of the product in the 64-bit integer pointed to by HighProduct and returns the low 64 bits of the product.
__mulh	Returns the high 64 bits of the product of two 64-bit signed integers.
__outbyte	Generates the **out** instruction, which sends 1 byte specified by Data out the I/O port specified by Port.
__outbytestring	Generates the **rep outsb** instruction,which sends the first Count bytes of data pointed to by Buffer to the port specified by Port.
__outdword	Generates the **out** instruction to send a doubleword Data out the port Port.
__outdwordstring	Generates the **rep outsd** instruction, which sends Count doublewords starting at Buffer out the I/O port specified by Port.
__rdtsc	Generates the **rdtsc** instruction, which returns the processor time stamp. The processor time stamp records the number of clock cycles since the last reset.
_ReadBarrier	Forces memory reads to complete.
__readcr0, __readcr2, __readcr3, __readcr4, __readcr8	Read the control registers. These intrinsics are only available in kernel mode.
__readfsbyte, __readfsdword, __readfsqword, __readfsword	Read memory from a location specified by an offset relative to the beginning of the FS segment. These intrinsics are only available in kernel mode.
__readgsbyte, __readgsdword, __readgsqword, __readgsword	Read memory from a location specified by an offset relative to the beginning of the GS segment. These intrinsics are only available in kernel mode.
__readmsr	Generates the **rdmsr** instruction, which reads the model-specific register specified by register and returns its value. This function may only be used in kernel mode.
__readpmc	Generates the **rdpmc** instruction, which reads the performance

	monitoring counter specified by counter.
_ReadWriteBarrier	Effectively blocks an optimization of reads and writes to global memory.
_ReturnAddress	The **_ReturnAddress** intrinsic provides the address of the instruction in the calling function that will be executed after control returns to the caller.
__shiftleft128	Shifts a 128-bit quantity, represented as two 64-bit quantities LowPart and HighPart, to the left by a number of bits specified by Shift and returns the high 64 bits of the result.
__shiftright128	Shifts a 128-bit quantity, represented as two 64-bit quantities LowPart and HighPart, to the right by a number of bits specified by Shift and returns the low 64 bits of the result.
__store128, __store128_rel	Stores a 128-bit value atomically.
__stosb	Generates a store string instruction (**rep stosb**).
__stosd	Generates a store string instruction (**rep stosd**).
__stosq	Generates a store string instruction (**rep stosq**).
__stosw	Generates a store string instruction (**rep stosw**).
__ull_rshift	on x64, shifts a 64-bit value specified by the first parameter to the right by a number of bits specified by the second parameter.
_umul128	Multiplies two 64-bit unsigned integers passed in as the first two arguments and puts the high 64 bits of the product in the 64-bit unsigned integer pointed to by HighProduct and returns the low 64 bits of the product.
__umulh	Return the high 64 bits of the product of two 64-bit unsigned integers.
__wbinvd	Generates the Write Back and Invalidate Cache (**wbinvd**) instruction.
_WriteBarrier	Forces memory writes to complete and be correct according to program logic at the point of the call.
__writecr0, __writecr3, __writecr4, __writecr8	Write the control registers. These intrinsics are only available in kernel mode.
__writefsbyte, __writefsdword, __writefsqword, __writefsword	Write memory to a location specified by an offset relative to the beginning of the FS segment. These intrinsics are only available in kernel mode.
__writegsbyte,	Write memory to a location specified by an offset relative to the

__writegsdword, __writegsqword, __writegsword	beginning of the GS segment. These intrinsics are only available in kernel mode.
__writemsr	Generates the Write to Model Specific Register (**wrmsr**) instruction. This function may only be used in kernel mode.

There are also some 3D intrinsics (called 3DNow) which will be useful for game/3D coders. I left those intrinsics out of the list since they were too many and you'd need to include another header file to use them: "mm3dnow.h".

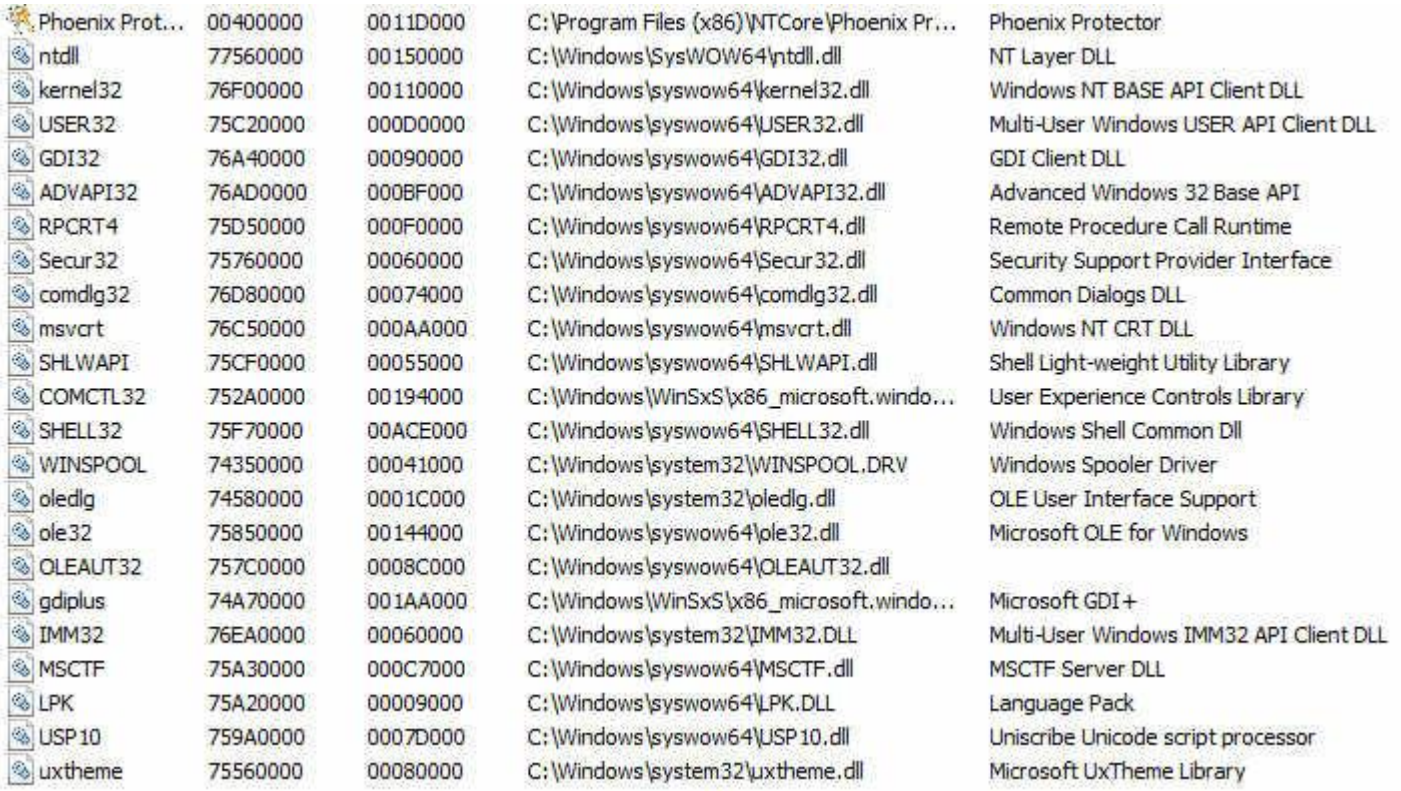
If these intrinsics are not enough, you might need to use an external asm file. On the other hand, if you're really lazy and you just need something on the fly, there's a quick way to embed assembly code in your C/C++ files.

```
#include "stdafx.h"
#include <Windows.h>

unsigned char BitSwapAsm[7] =
{
    0x48, 0x8B, 0xC1, // mov rax, rcx
    0x48, 0x0F, 0xC8, // bswap rax
    0xC3              // retn
};
__int64 (*BitSwap)(__int64 Value) = (__int64 (*)(__int64)) (ULONG_PTR) BitSwapAsm;

int _tmain(int argc, _TCHAR* argv[])
{
    //
    // I have to change the page protection, otherwise the code would crash
    //
    DWORD dwOldProtect;
    VirtualProtect(BitSwap, sizeof (BitSwapAsm), PAGE_EXECUTE_READWRITE,
&dwOldProtect);

    printf_s("%p\n", BitSwap(0xDDCCBBAA));
    getchar();
}
```

This code relies on function pointers and I had to change the page protection flags in order to make it execute. It's really a dumb method, but in some case it could be time saving.

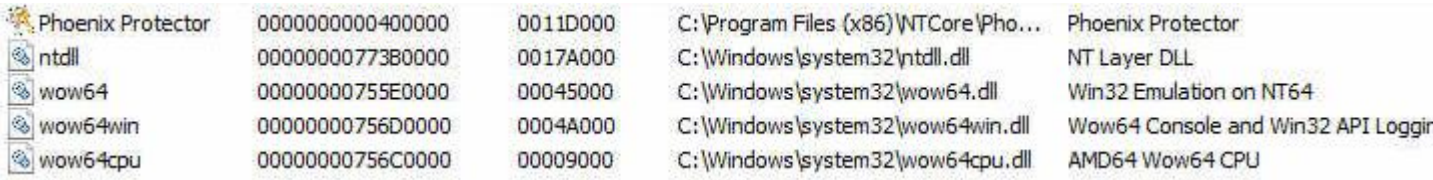
Windows On Windows

Of course, compatibility for 32bit applications has to be provided on x64 (and Itanium as well) and this is what WOW64 (Windows on Windows 64) is all about. When we look at the modules loaded by a 32bit application with a 32bit version of the Task Explorer we see this:



Phoenix Prot...	00400000	0011D000	C:\Program Files (x86)\NTCore\Phoenix Pr...	Phoenix Protector
ntdll	77560000	00150000	C:\Windows\SysWOW64\ntdll.dll	NT Layer DLL
kernel32	76F00000	00110000	C:\Windows\syswow64\kernel32.dll	Windows NT BASE API Client DLL
USER32	75C20000	000D0000	C:\Windows\syswow64\USER32.dll	Multi-User Windows USER API Client DLL
GDI32	76A40000	00090000	C:\Windows\syswow64\GDI32.dll	GDI Client DLL
ADVAPI32	76AD0000	000BF000	C:\Windows\syswow64\ADVAPI32.dll	Advanced Windows 32 Base API
RPCRT4	75D50000	000F0000	C:\Windows\syswow64\RPCRT4.dll	Remote Procedure Call Runtime
Secur32	75760000	00060000	C:\Windows\syswow64\Secur32.dll	Security Support Provider Interface
comdlg32	76D80000	00074000	C:\Windows\syswow64\comdlg32.dll	Common Dialogs DLL
msvcrt	76C50000	000AA000	C:\Windows\syswow64\msvcrt.dll	Windows NT CRT DLL
SHLWAPI	75CF0000	00055000	C:\Windows\syswow64\SHLWAPI.dll	Shell Light-weight Utility Library
COMCTL32	752A0000	00194000	C:\Windows\WinSxS\x86_microsoft.windo...	User Experience Controls Library
SHELL32	75F70000	00ACE000	C:\Windows\syswow64\SHELL32.dll	Windows Shell Common Dll
WINSPOOL	74350000	00041000	C:\Windows\system32\WINSPOOL.DRV	Windows Spooler Driver
oledlg	74580000	0001C000	C:\Windows\system32\oledlg.dll	OLE User Interface Support
ole32	75850000	00144000	C:\Windows\syswow64\ole32.dll	Microsoft OLE for Windows
OLEAUT32	757C0000	0008C000	C:\Windows\syswow64\OLEAUT32.dll	
gdiplus	74A70000	001AA000	C:\Windows\WinSxS\x86_microsoft.windo...	Microsoft GDI+
IMM32	76EA0000	00060000	C:\Windows\system32\IMM32.DLL	Multi-User Windows IMM32 API Client DLL
MSCTF	75A30000	000C7000	C:\Windows\syswow64\MSCTF.dll	MSCTF Server DLL
LPK	75A20000	00009000	C:\Windows\syswow64\LPK.DLL	Language Pack
USP10	759A0000	0007D000	C:\Windows\syswow64\USP10.dll	Uniscribe Unicode script processor
uxtheme	75560000	00080000	C:\Windows\system32\uxtheme.dll	Microsoft UxTheme Library

Seems pretty regular, except, of course, for the system files path, which in our case is syswow64 instead of the old common System32. It's easy to understand why it is this way: the System32 folder is now reserved for the 64bit environment and the 32bit files had to be placed somewhere else. But look what happens when I open the same process with an x64 version of the Task Explorer:



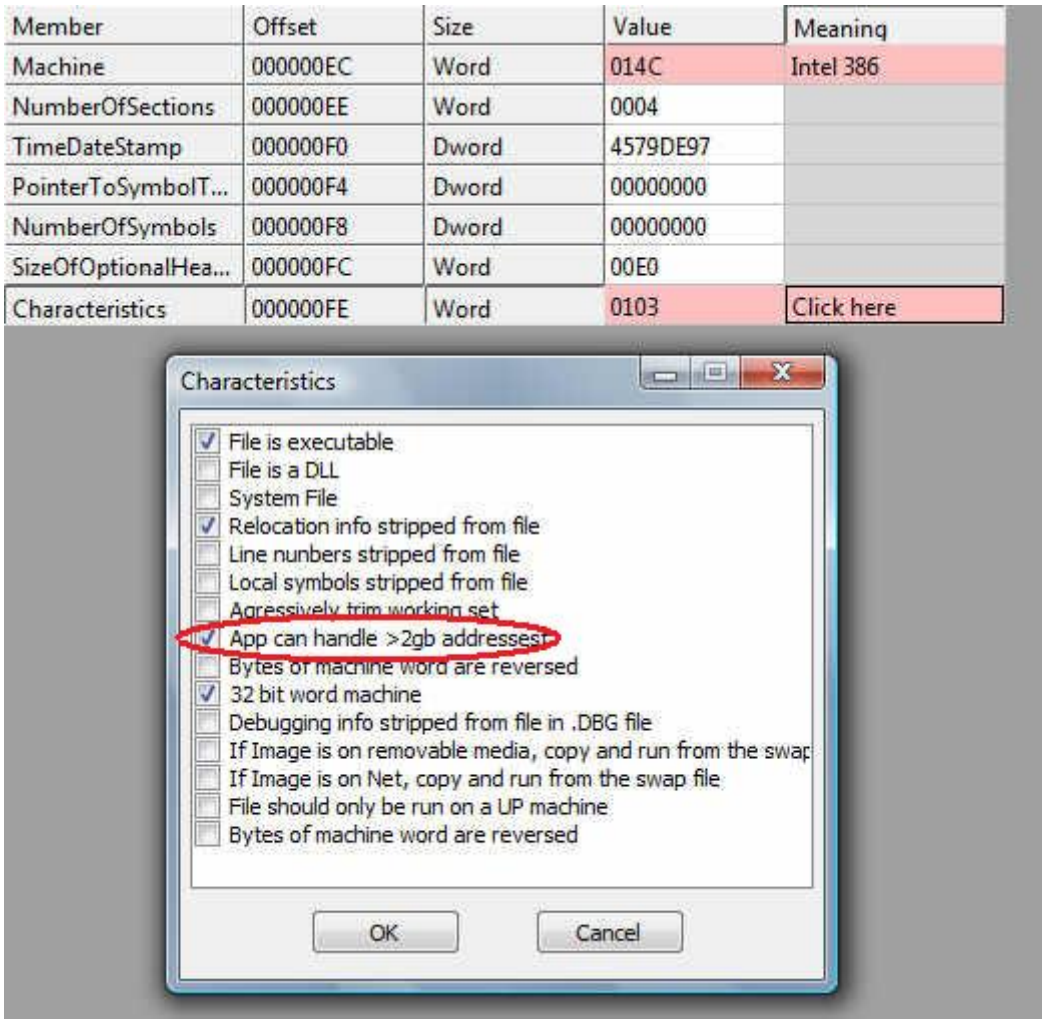
Phoenix Protector	0000000000400000	0011D000	C:\Program Files (x86)\NTCore\Pho...	Phoenix Protector
ntdll	00000000773B0000	0017A000	C:\Windows\system32\ntdll.dll	NT Layer DLL
wow64	00000000755E0000	00045000	C:\Windows\system32\wow64.dll	Win32 Emulation on NT64
wow64win	00000000756D0000	0004A000	C:\Windows\system32\wow64win.dll	Wow64 Console and Win32 API Loggir
wow64cpu	00000000756C0000	00009000	C:\Windows\system32\wow64cpu.dll	AMD64 Wow64 CPU

Suddenly, all the 32bit modules are gone and what remains are the WOW64 emulation modules. Here's the description the MSDN gives us of these modules:

> The WOW64 emulator runs in user mode, provides an interface between the 32-bit version of Ntdll.dll and the kernel of the processor, and it intercepts kernel calls. The emulator consists of the following DLLs:
>
> - Wow64.dll provides the core emulation infrastructure and the thunks for the Ntoskrnl.exe entry-point functions.
> - Wow64Win.dll provides thunks for the Win32k.sys entry-point functions.
> - Wow64Cpu.dll provides x86 instruction emulation on Itanium processors. It executes mode-switch instructions on the processor. This DLL is not necessary for x64 processors because they execute x86-32 instructions at full clock speed.
>
> Along with the 64-bit version of Ntdll.dll, these are the only 64-bit binaries that can be loaded into a 32-bit process.At startup, Wow64.dll loads the x86 version of Ntdll.dll and runs its initialization code, which loads all necessary 32-bit DLLs. Almost all 32-bit DLLs are unmodified copies of 32-bit Windows binaries. However, some of these DLLs are written to behave differently on WOW64 than they do on 32-bit Windows [...].
>
> Instead of using the x86 system-service call sequence, 32-bit binaries that make system calls are rebuilt to use a custom calling sequence. This new sequence is inexpensive for WOW64 to intercept because it remains entirely in user mode. When the new calling sequence is detected, the WOW64 CPU transitions back to native 64-bit mode and calls into Wow64.dll. Thunking is done in user mode to reduce the impact on the 64-bit kernel, and to reduce the risk of a bug in the thunk that causes

a kernel-mode crash, data corruption, or a security hole. The thunks extract arguments from the 32-bit stack, extend them to 64 bits, then make the native system call.

32bit applications have a maximal 2GB space (4GB if explicitly required) and the rest of the space is handled by the system. This doesn't change much of course, since on x86 user mode applications had 2GB of virtual memory space out of 4GB (the other 2GB were reserved for kernel mode). On x64 these two other GB can now be accessed by 32bit applications. In order to achieve this, the IMAGE_FILE_LARGE_ADDRESS_AWARE flag has to be set in the File Header's Characteristics field. You can do this programmatically or manually with a normal PE editor like the CFF Explorer, just like this:



I've seen this done by 3D-games players in order to increase performances. Of course, it's only useful for very heavy memory consuming applications.

A very useful function to determine whether a process is running under WOW64 or not is:

```
BOOL IsWow64Process(
    HANDLE hProcess,        // [in] Handle to a process.
    PBOOL Wow64Process      // [out] Pointer to a value that is set to TRUE if the
process is
                            // running under WOW64. Otherwise, the value is set to
FALSE.
);
```

The work done by Wow64Cpu.dll on x64 is zero, because x64 supports x86 natively. I was first tempted to look how the calling sequence works in order to make one myself and provide a way to use x86 components from x64 in the same address space, but, on second thought, even if it could be implemented, it wouldn't work on Itanium. And this brings us to one of the next paragraphs, because under normal conditions a 32bit application cannot load a 64bit dll and a 64bit application cannot load a 32bit dll. So, interprocess communication becomes an important aspect on 64bit systems. Anyway, before that, I have to talk about file system and registry redirection, since they

are strictly related to WOW64, but deserve an extra paragraph for their importance.

File System And Registry Redirection

Since the System32 path is reserved to 64bit files, any time a 32bit application tries to access this directory it is redirected to SysWow64 one. However, there are some subdirectories of System32 that are shared between 32bit and 64bit applications and so no redirection is needed. These subdirectories are:

- %windir%\system32\catroot
- %windir%\system32\catroot2
- %windir%\system32\drivers\etc
- %windir%\system32\logfiles
- %windir%\system32\spool

Also, there are some functions related to the WOW64 file system redirection:

GetSystemWow64Directory	Retrieves the path of the system directory used by WOW64. This directory is not present on 32-bit Windows.
Wow64DisableWow64FsRedirection	Disables file system redirection for the calling thread. File system redirection is enabled by default.
Wow64EnableWow64FsRedirection	Enables or disables file system redirection for the calling thread. This function may not work reliably when there are nested calls. Therefore, this function has been replaced by the Wow64DisableWow64FsRedirection and Wow64RevertWow64FsRedirection functions.
Wow64RevertWow64FsRedirection	Restores file system redirection for the calling thread.

I think it's easy to understand how to use these functions. However, I add a little code sample (you can find almost the same one on the MSDN):

```
int _tmain(int argc, _TCHAR* argv[])
{
   BOOL bIsWOW64Enabled;

   if (IsWow64Process(GetCurrentProcess(), &bIsWOW64Enabled))
   {
      if (bIsWOW64Enabled == TRUE) // we run under WOW64
      {
         PVOID pOldValue;
         DWORD FileSize;

         HANDLE hFile = CreateFile(_T("c:\\windows\\system32\\notepad.exe"),
GENERIC_READ,
            FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);

         FileSize = GetFileSize(hFile, NULL);

         CloseHandle(hFile);

         _tprintf(_T("File Size: %d Bytes\n"), FileSize);

         Wow64DisableWow64FsRedirection(&pOldValue);  // disable redirection

         hFile = CreateFile(_T("c:\\windows\\system32\\notepad.exe"), GENERIC_READ,
            FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);
```

```
            FileSize = GetFileSize(hFile, NULL);

            CloseHandle(hFile);

            _tprintf(_T("File Size: %d Bytes\n"), FileSize);

            Wow64RevertWow64FsRedirection(pOldValue);  // restore redirection

            getchar();
        }
    }

    return 0;
}
```

The output of this program is:

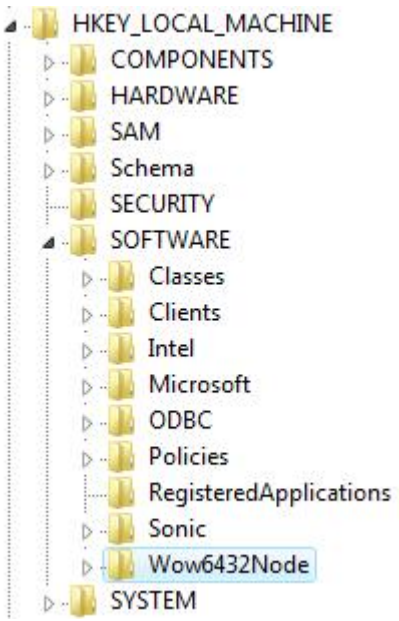
```
File Size: 151040
Bytes
File Size: 169472
Bytes
```

The file size changes because one time the program opens the 32bit notepad and one time the 64bit one. Of course, remember when you're using these functions, always use them along with GetProcAddress, otherwise your code won't work on older systems which don't provide them.

Let's move on to the registry. As for the file system the registry is being redirected as well, or better some keys of it. These keys are:

- HKEY_LOCAL_MACHINE\Software
- HKEY_USERS*\Software\Classes
- HKEY_USERS*_Classes

You can find every one of these keys duplicated for 32bit applications in their WOW node: any of these keys has a subkey called Wow6432Node, which contains a duplicate of the parent key. For instance:



Some of these WOW64 redirected keys have subkeys which are reflected. Reflection in this case means that when I change a reflected key in the 32bit node the change is being *reflected* on the 64bit key as well and viceversa. This is necessary, because some keys need to remain in synch. This is quite different from just sharing the keys between 64bit and 32bit mode, because the reflection can be filtered and also disabled. These are the reflected keys:

- HKEY_LOCAL_MACHINE\Software\Classes
- HKEY_LOCAL_MACHINE\Software\Microsoft\COM3
- HKEY_LOCAL_MACHINE\Software\Microsoft\EventSystem
- HKEY_LOCAL_MACHINE\Software\Microsoft\Ole
- HKEY_LOCAL_MACHINE\Software\Microsoft\Rpc
- HKEY_USERS*\Software\Classes
- HKEY_USERS*_Classes

The functions to handle reflection are:

RegQueryReflectionKey	Determines whether reflection has been disabled or enabled for the specified key.
RegDisableReflectionKey	Disables registry reflection for the specified key. Disabling reflection for a key does not affect reflection of any subkeys.
RegEnableReflectionKey	Restores registry reflection for the specified disabled key. Restoring reflection for a key does not affect reflection of any subkeys.

They work just like the WOW64 file system functions, so I don't think a code sample is necessary. There are also some shared keys between 64bit and 32bit applications:

- HKEY_LOCAL_MACHINE\SOFTWARE\Classes\HCP
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography\Calais\Current
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography\Calais\Readers
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography\Services
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\CTF\SystemShared
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\CTF\TIP
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\DFS
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Driver Signing
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\EnterpriseCertificates
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSMQ
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Non-Driver Signing
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\RAS
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Software\Microsoft\Shared Tools\MSInfo
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\SystemCertificates
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\TermServLicensing
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Transaction Server
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\FontDpi
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\FontMapper
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Fonts
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\FontSubstitutes
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\NetworkCards
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Perflib
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Ports
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Print
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\ProfileList
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Time Zones
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Control Panel\Cursors\Schemes
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Group Policy
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Setup
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Setup\OC Manager
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Telephony\Locations
- HKEY_LOCAL_MACHINE\SOFTWARE\Policies

As said, these keys are shared, so any change made to them will affect both 32bit and 64bit applications, and there's no way to avoid this like for reflected keys.

But what if a 32bit applications wants to access the 64bit registry or viceversa? Don't worry! As I

discovered when I was dealing with the same problem, Microsoft provides a very simple way to do the job. The flags KEY_WOW64_64KEY and KEY_WOW64_32KEY can be used with these functions: RegCreateKeyEx, RegDeleteKeyEx and RegOpenKeyEx.

KEY_WOW64_64KEY	Access a 64-bit key from either a 32-bit or 64-bit application.
KEY_WOW64_32KEY	Access a 32-bit key from either a 32-bit or 64-bit application.

What I needed to do was to access the subkeys of a 64bit key from a 32bit application, which translated in code is just:

```
RegOpenKeyEx(HKEY_LOCAL_MACHINE, MyKey, 0, KEY_READ | KEY_WOW64_64KEY, &hKey);
```

Easy, isn't it?

All in all, the documentation provided by Microsoft on file system and registry redirection is very good and I just reported what I first found on the MSDN. I don't think these redirections are going to be much of a problem for programmers.

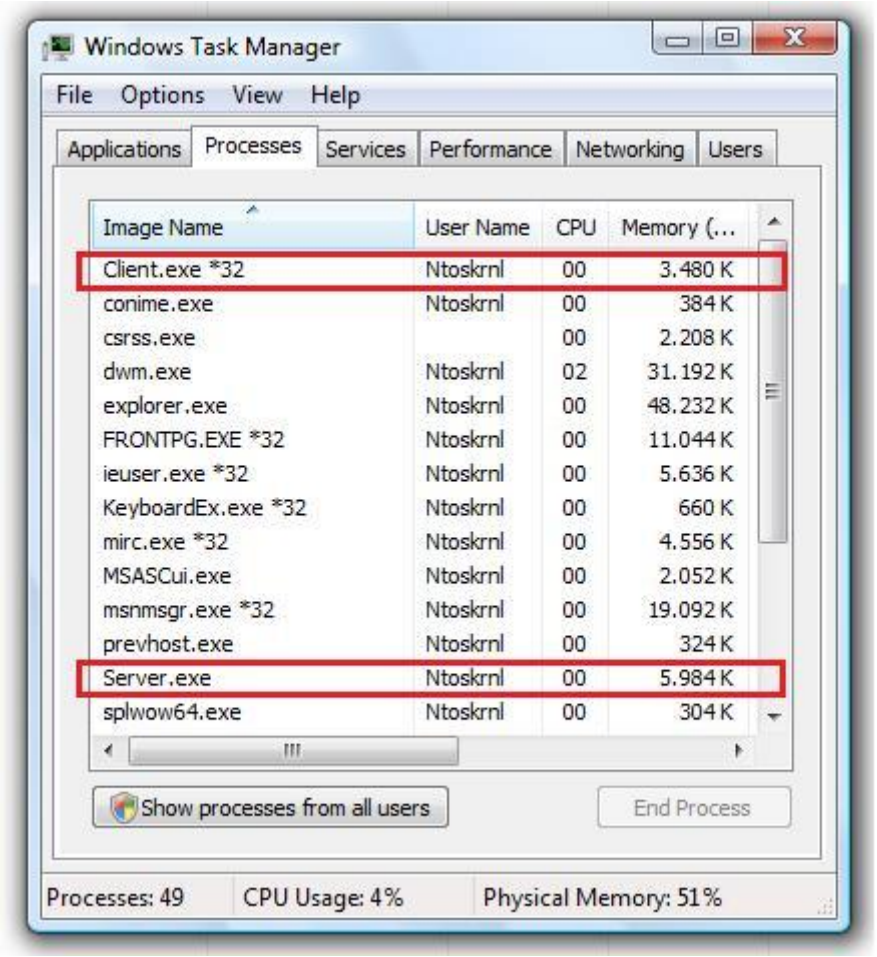
Interprocess Communication

As mentioned in the Windows On Windows paragraph, interprocess communication becomes an important aspect on x64, since a 64bit application might need to use a 32bit component and viceversa. The MSDN suggests these ways for process to communicate between each other:

- Handles to named objects such as mutexes, semaphores, and file handles can all be shared.
- Handles to windows (HWND) can be shared.
- RPC.
- COM LocalServers.
- Shared memory can be used if the contents of the shared memory are not pointer-dependent.
- The CreateProcess and ShellExecute functions can launch 32-bit and 64-bit processes from either 32-bit or 64-bit processes.
- The CreateRemoteThread function is special-cased for specific functions, allowing 64-bit debuggers to break into 32-bit processes.

Using CreateProcess or ShellExecute means that you could communicate through arguments and output reading. If you need something more sofisticated (and professional), you have no choice but to use RPCs (Remote Procedure Calls) or COM objects. For RPCs you need to learn a bit about the MIDL (Microsoft Interface Definition Language), but eventually every code sample I tried wasn't working on Vista x64, so I gave up on RPCs. I would suggest you to use a COM, writing them in MFC is very easy (comparing to writing them without MFC, I mean). There's a very good series of articles on CodeProject about writing ActiveXs. Actually, the guide is about how writing ActiveXs in plain C (I had to reduce the size of my ActiveX, so I couldn't use MFC), but the theory is the same and these articles are well written and could save you from the effort of reading a book. If you have never written COM objects before, you will eventually discover that it can be annoying.

Shared memory is not really an option. If you are looking for a solution between CreateProcess and COM objects, you may use pipes or things like that. Actually, you could implement your own pipes through shared memory and mutexes. This is what I have done in some projects:



The " *32" next to the process name is the way of the Task Manager to tell us which are 32bit processes. As you can see the Server is a 64bit process and the Client a 32bit one. The two processes communicate with each other without problems. However, don't get too excited, there are some problems and I'll explain later what they are about. For now, let's see a code sample.

Open Communication.zip (22kb) from “Files” directory inside the package.

Here's the Client code:

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

#define BUF_SIZE  256 * sizeof (TCHAR)

TCHAR MyEvent[] = _T("Global\\SharedMemoryEvent");

TCHAR szName[]= _T("Global\\MyFileMappingObject");

int WINAPI _tWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR szCmdLine,
int iCmdShow)
{
   //
   // Create the event to communicate between server and client
   //

   HANDLE hEvent = CreateEvent(NULL, FALSE, FALSE, MyEvent);

   //
   // Start server process
   //

   PROCESS_INFORMATION pi = { 0 };
   STARTUPINFO si = { 0 };

   if (!CreateProcess(_T("Server.exe"), NULL, NULL, NULL, FALSE, 0, NULL, NULL, &si,
```

```
&pi))
       return 1;

   //
   // Wait for the server to complete the job
   //

   WaitForSingleObject(hEvent, INFINITE);

   //
   // Access shared memory object
   //

   HANDLE hMapFile = OpenFileMapping(
      FILE_MAP_ALL_ACCESS,   // read/write access
      FALSE,                 // do not inherit the name
      szName);               // name of mapping object

   if (hMapFile == NULL) return 1;

   LPCTSTR pBuf = (LPTSTR) MapViewOfFile(
      hMapFile,              // handle to map object
      FILE_MAP_ALL_ACCESS,   // read/write permission
      0,
      0,
      BUF_SIZE);

   if (pBuf == NULL) return 1;

   //
   // Shows Server Output
   //

   MessageBox(NULL, pBuf, _T("Server Output"), MB_OK);

   UnmapViewOfFile(pBuf);

   CloseHandle(hMapFile);

   //
   // Tell the server that the object isn't used any longer
   //

   SetEvent(hEvent);

   return 0;
}
```

And here's the Server code:

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

#define BUF_SIZE  256 * sizeof (TCHAR)

TCHAR MyEvent[] = _T("Global\\SharedMemoryEvent");

TCHAR szName[] = _T("Global\\MyFileMappingObject");
TCHAR szMsg[] = _T("Message from server process");

int WINAPI _tWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR szCmdLine,
int iCmdShow)
{
   //
   // Create the memory shared object
   //

   HANDLE hMapFile;
```

```
    LPCTSTR pBuf;

    hMapFile = CreateFileMapping(
       INVALID_HANDLE_VALUE,    // use paging file
       NULL,                    // default security
       PAGE_READWRITE,          // read/write access
       0,                       // max. object size
       BUF_SIZE,                // buffer size
       szName);                 // name of mapping object

    if (hMapFile == NULL) return 1;

    pBuf = (LPTSTR) MapViewOfFile(
       hMapFile,            // handle to map object
       FILE_MAP_ALL_ACCESS, // read/write permission
       0,
       0,
       BUF_SIZE);

    if (pBuf == NULL) return 1;

    CopyMemory((PVOID) pBuf, szMsg, (_tcslen(szMsg) + 1) * sizeof (TCHAR));

    //
    // Wait for event before closing file object
    //

    HANDLE hEvent = OpenEvent(EVENT_ALL_ACCESS, FALSE, MyEvent);

    SetEvent(hEvent);

    WaitForSingleObject(hEvent, INFINITE);

    UnmapViewOfFile(pBuf);
    CloseHandle(hMapFile);

    return 0;
}
```

What these applications do is:

- The client creates a communication event.
- The client starts the server and waits for the communication event to be set.
- The server creates a shared memory object and fills it with an output.
- The server sets the communication event in order to tell the client to process the output.
- The server waits for the client to clear the shared memory.
- The client processes the server's output.
- The client tells the server that it can now clear the shared memory.

I believe it's easier to understand the code itself than this list. The problem I mentioned earlier is that in order to share a memory object (or an event) between processes, I have to create it in the "Global*" section. What happens with Vista is that only applications with admin privileges can access this section with CreateFileMapping (no problems with mutexes or events, though), and since usually applications run in Vista with user privileges, you have to explicitly tell Vista to run the Client application with admin privileges, which is not very professional. The solution to this problem could be to share the memory through a temporary file or even the registry (for small data).

Portable Executable

If your software has anything to do with Portable Executables it won't be too hard to move to x64 (if you haven't done it already). Basically, what in PE64 changes is the size of virtual addresses (VAs), which are now 64bit wide. Keep in mind that not all the fields described as virtual addresses really are such, most of the time they're just relative virtual addresses (RVAs), which are, like in the PE32, 32bit wide. What changes, in short, is the Optional Header (which has some 64bit wide fields like the ImageBase), Import Directory thunks (the two thunk arrays. OFTs and FTs, are now 64bit wide, since thunks were built to contain virtual addresses among the other things), the Load Config Directory and the TLS Directory.

Let's take, for instance, the old PE32 Optional Header:

```
typedef struct _IMAGE_OPTIONAL_HEADER {
    //
    // Standard fields.
    //

    WORD    Magic;
    BYTE    MajorLinkerVersion;
    BYTE    MinorLinkerVersion;
    DWORD   SizeOfCode;
    DWORD   SizeOfInitializedData;
    DWORD   SizeOfUninitializedData;
    DWORD   AddressOfEntryPoint;
    DWORD   BaseOfCode;
    DWORD   BaseOfData;

    //
    // NT additional fields.
    //

    DWORD   ImageBase;
    DWORD   SectionAlignment;
    DWORD   FileAlignment;
    WORD    MajorOperatingSystemVersion;
    WORD    MinorOperatingSystemVersion;
    WORD    MajorImageVersion;
    WORD    MinorImageVersion;
    WORD    MajorSubsystemVersion;
    WORD    MinorSubsystemVersion;
    DWORD   Win32VersionValue;
    DWORD   SizeOfImage;
    DWORD   SizeOfHeaders;
    DWORD   CheckSum;
    WORD    Subsystem;
    WORD    DllCharacteristics;
    DWORD   SizeOfStackReserve;
    DWORD   SizeOfStackCommit;
    DWORD   SizeOfHeapReserve;
    DWORD   SizeOfHeapCommit;
    DWORD   LoaderFlags;
    DWORD   NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

And the PE64 one:

```
typedef struct _IMAGE_OPTIONAL_HEADER64 {
    WORD        Magic;
    BYTE        MajorLinkerVersion;
    BYTE        MinorLinkerVersion;
    DWORD       SizeOfCode;
    DWORD       SizeOfInitializedData;
    DWORD       SizeOfUninitializedData;
    DWORD       AddressOfEntryPoint;
```

```
    DWORD       BaseOfCode;
    ULONGLONG   ImageBase;
    DWORD       SectionAlignment;
    DWORD       FileAlignment;
    WORD        MajorOperatingSystemVersion;
    WORD        MinorOperatingSystemVersion;
    WORD        MajorImageVersion;
    WORD        MinorImageVersion;
    WORD        MajorSubsystemVersion;
    WORD        MinorSubsystemVersion;
    DWORD       Win32VersionValue;
    DWORD       SizeOfImage;
    DWORD       SizeOfHeaders;
    DWORD       CheckSum;
    WORD        Subsystem;
    WORD        DllCharacteristics;
    ULONGLONG   SizeOfStackReserve;
    ULONGLONG   SizeOfStackCommit;
    ULONGLONG   SizeOfHeapReserve;
    ULONGLONG   SizeOfHeapCommit;
    DWORD       LoaderFlags;
    DWORD       NumberOfRvaAndSizes;
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER64, *PIMAGE_OPTIONAL_HEADER64;
```

Of course, ULONGLONG are 64bit wide fields. As you can see, the AddressOfEntryPoint remains, as every RVA, a dword. Oppositely, ImageBase, being a Virtual Address, becomes a qword.

Distinguishing between PE32 and PE64 should be done by checking the Magic field in the Optional Header. This field can be one of these values:

```
#define IMAGE_NT_OPTIONAL_HDR32_MAGIC      0x10b
#define IMAGE_NT_OPTIONAL_HDR64_MAGIC      0x20b
#define IMAGE_ROM_OPTIONAL_HDR_MAGIC       0x107
```

It is your choice to either double write every time the code to handle both PE32/64 or write a class to handle them automatically.

Exception Handling

Remember the old days when you set the SEH in your code? Well, with x64/Itanium they're gone. Exception Handlers are now stored as structured in the PE64 Exception Directory. The basic structure is this:

```
typedef struct _RUNTIME_FUNCTION {
    ULONG BeginAddress;
    ULONG EndAddress;
    ULONG UnwindData;
} RUNTIME_FUNCTION, *PRUNTIME_FUNCTION;
```

All three fields are RVAs (otherwise there wouldn't be dwords).

BeginAddress	Points to the start address of the involved part of code.
EndAddress	Points to the end address of the same part of code.
UnwindData	Points to an UNWIND_INFO structure.

The UNWIND_INFO structure tells how the portion of code should be handled. Here's the declaration I found on MSDN:

```
typedef union _UNWIND_CODE {
    struct {
        UBYTE CodeOffset;
        UBYTE UnwindOp : 4;
        UBYTE OpInfo   : 4;
    };
    USHORT FrameOffset;
} UNWIND_CODE, *PUNWIND_CODE;

typedef struct _UNWIND_INFO {
    UBYTE Version       : 3;
    UBYTE Flags         : 5;
    UBYTE SizeOfProlog;
    UBYTE CountOfCodes;
    UBYTE FrameRegister : 4;
    UBYTE FrameOffset   : 4;
    UNWIND_CODE UnwindCode[1];
/*  UNWIND_CODE MoreUnwindCode[((CountOfCodes + 1) & ~1) - 1];
*   union {
*       OPTIONAL ULONG ExceptionHandler;
*       OPTIONAL ULONG FunctionEntry;
*   };
*   OPTIONAL ULONG ExceptionData[]; */
} UNWIND_INFO, *PUNWIND_INFO;
```

Here's the description of the UNWIND_INFO structure members taken directly from the MSDN:

Version	Version number of the unwind data, currently 1.
Flags	Three flags are currently defined: UNW_FLAG_EHANDLER The function has an exception handler that should be called when looking for functions that need to examine exceptions. UNW_FLAG_UHANDLER The function has a termination handler that should be called when unwinding an exception. UNW_FLAG_CHAININFO This unwind info structure is not the primary one for the procedure. Instead, the chained unwind info entry is the contents of a previous RUNTIME_FUNCTION entry. See the following text for an explanation of chained unwind info structures. If this flag is set, then the UNW_FLAG_EHANDLER and UNW_FLAG_UHANDLER flags must be cleared. Also, the frame register and fixed-stack allocation fields must have the same values as in the primary unwind info.
SizeOfProlog	Length of the function prolog in bytes.
CountOfCodes	This is the number of slots in the unwind codes array. Note that some unwind codes (for example, UWOP_SAVE_NONVOL) require more than one slot in the array.
FrameRegister	If nonzero, then the function uses a frame pointer, and this field is the number of the nonvolatile register used as the frame pointer, using the same encoding for the operation info field of UNWIND_CODE nodes.
FrameOffset	If the frame register field is nonzero, then this is the scaled offset from RSP that is applied to the FP reg when it is established. The actual FP reg is set to RSP + 16 * this number, allowing offsets from 0 to 240. This permits pointing the FP reg into the middle of the local stack allocation for dynamic stack frames, allowing better code

	density through shorter instructions (more instructions can use the 8-bit signed offset form).
UnwindCode	This is an array of items that explains the effect of the prolog on the nonvolatile registers and RSP. See the section on UNWIND_CODE for the meanings of individual items. For alignment purposes, this array will always have an even number of entries, with the final entry potentially unused (in which case the array will be one longer than indicated by the count of unwind codes field).
ExceptionHandler	This is an image-relative pointer to either the function's language-specific exception/termination handler (if flag UNW_FLAG_CHAININFO is clear and one of the flags UNW_FLAG_EHANDLER or UNW_FLAG_UHANDLER is set).
Language-specific handler data (ExceptionData)	This is the function's language-specific exception handler data. The format of this data is unspecified and completely determined by the specific exception handler in use.
Chained Unwind Info (ExceptionData)	If flag UNW_FLAG_CHAININFO is set then the UNWIND_INFO structure ends with three UWORDs. These UWORDs represent the RUNTIME_FUNCTION information for the function of the chained unwind.

The possible values of the Flags field are:

```
#define UNW_FLAG_EHANDLER  0x01
#define UNW_FLAG_UHANDLER  0x02
#define UNW_FLAG_CHAININFO 0x04
```

Let's take for instance this code:

```
#include <Windows.h>
#include <intrin.h>

int APIENTRY _tWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                       LPTSTR lpCmdLine, int nCmdShow)
{
   __try
   {
      __debugbreak();
   }
   __except (EXCEPTION_EXECUTE_HANDLER)
   {
      MessageBox(0, _T("Hello!"), _T("SEH"), MB_OK);
   }

   return 0;
}
```

The dissassembly would be:

```
.text:0000000000401000 wWinMain proc near          ; CODE XREF: __tmainCRTStartup+18C p
.text:0000000000401000    sub rsp, 28h             ; BeginAddress
.text:0000000000401004    int 3                    ; Trap to Debugger
.text:0000000000401005    jmp short loc_401021
.text:0000000000401007 ;
-------------------------------------------------------------------------------
.text:0000000000401007    xor r9d, r9d             ; ExceptionHandler
.text:000000000040100A    lea r8, Caption
.text:0000000000401011    lea rdx, Text
.text:0000000000401018    xor ecx, ecx
```

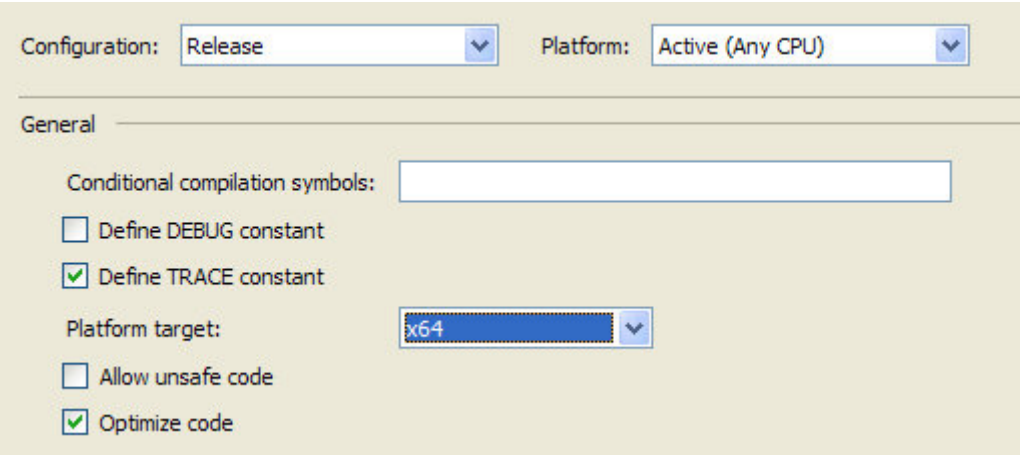
```
.text:000000000040101A     call cs:__imp_MessageBoxW
.text:0000000000401020     nop
.text:0000000000401021
.text:0000000000401021 loc_401021:                 ; CODE XREF: wWinMain+5 j
.text:0000000000401021     xor eax, eax
.text:0000000000401023     add rsp, 28h
.text:0000000000401027     retn
.text:0000000000401027 wWinMain endp               ; EndAddress (+ alignment)
```

If you need to generate code dinamically and set for it an exception handler, you can use the function RtlAddFunctionTable, which takes as paramater an array of RUNTIME_FUNCTION structures. This means, of course, that you'll have to fill one or more UNWIND_INFO structure/s by yourself. It's certainly a bit more complicated than on x86, but my guess is that a lot of software protections are going to use this method.

.NET Framework

Both the x86 and x64 .NET frameworks can coexist peacefully on a x64 Windows. In case you install both, there will be two things of everything: two directories in the .NET directory, two directories in the global cache and two main registry keys. Since .NET assemblies don't contain native code, why could it be useful to have both frameworks on the same computer?

.NET assemblies can call native code through the *System.Runtime.InteropServices* namespace. Of course, a .NET assembly which runs on the x64 .NET framework is a 64bit process, therefore it can't call x86 components functions. Viceversa, assemblies executed on x86 can't use x64 components. You can explicitly tell the Visual Studio to compile your assembly for a specific platform (x86, x64, Itanium) by just going on Project -> Properties -> Build.



64bit PEs are always built for a specific platform, only PE32 assemblies are allowed to run on every framework (x86 systems wouldn't be able to execute a PE64). Anyway, it's possible to make PE32 assemblies run just on the x86 framework by just setting one flag in the .NET Directory (IMAGE_COR20_DIRECTORY). The flag is COMIMAGE_FLAGS_32BITREQUIRED. By setting this flag you'll force to execute the given assembly as a 32bit process even on 64bit platforms.

There are also some differences between the 32bit and 64bit .NET framework. I noticed that the 64bit one is very serious about alignments and integrity cheks in assemblies, and the new 3.0 framework has even more checks.

Vista Section

Editions

Windows Vista is shipped in various editions, before you install the new system, you should be aware of the features missing in some editions. Here's the official features table given by Microsoft:

Features	Home	Premium	Business	Ultimate
Most secure Windows ever with Windows Defender and Windows Firewall	✓	✓	✓	✓
Quickly find what you need with Instant Search and Windows Internet Explorer 7	✓	✓	✓	✓
Elegant Windows Aero desktop experience with Windows Flip 3D navigation		✓	✓	✓
Best choice for laptops with enhanced Windows Mobility Center and Tablet PC support		✓	✓	✓
Collaborate and share documents with Windows Meeting Space		✓	✓	✓
Experience photos and entertainment in your living room with Windows Media Center		✓		✓
Enjoy Windows Media Center on TVs throughout your home with Xbox 360™ and other devices		✓		✓
Help protect against hardware failure with advanced business backup features			✓	✓
Business networking and Remote Desktop for easier connectivity			✓	✓
Better protect your data against loss or theft with Windows BitLocker™ Drive Encryption				✓

But there are other things that the common user might not notice. For instance, the Business and Ultimate edition allow virtualization (using the system as a virtual machine). The End-User License Agreement for the Vista Home Basic and Premium reads:

4. **USE WITH VIRTUALIZATION TECHNOLOGIES.** You may not use the software installed on the licensed device within a virtual (or otherwise emulated) hardware system.

Apparently, Microsoft put special checks in the home editions, in order to prevent them from working in emulation. I read that VMWare was quite disappointed by this policy adopted by Microsoft. This controversy might change things, but I think many programmers and companies should know this before buying one or another Vista edition.

Microsoft Visual Studio

I chose this as the second paragraph of the Windows Vista section, because what every programmer first does when he has a brand new system is to install and set up his compilers. The problem is, since many things change with Vista, the only compatible Visual Studio platform is the 2005 one. Compatibility for Visual Studio 6 and Visual Studio .NET 2003 is no longer provided (although Visual Basic 6 seems to be supported). Not only that, in order to make the VS.NET 2005 work, you'll need to download and install the Service Pack 1 for it.

For a lot of programmers like myself it's not a big deal, since maintaining the code up to date is very important, but who has ever worked with little companies knows that a lot of them have no interest in doing the same, with the result that, for instance, many solutions are still developed for the .NET framework 1.0. Those solutions will, of course, still run on Vista, but there won't be any tool to compile them. I don't think Microsoft is going to solve this issue. Thus, for many companies it will take some time to switch to Vista.

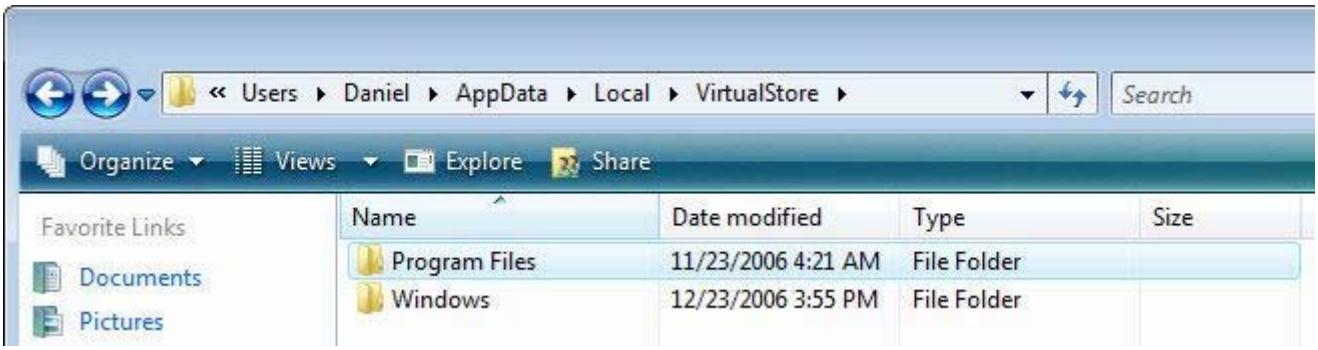
P.S. The setup of Microsoft Visual Studio's Service Pack 1 will take **much** time. Don't worry, it's normal.

User Account Control

What stands in the way of working properly for most applications on Windows Vista is the User Account Control (UAC), also known as Limited User Access (LUA). As we saw in the Interprocess Communication paragraph, admin rights to create shared memory objects are necessary, but Windows Vista runs every process (except system processes) with user rights. Incompatibilities are, most of the times, generated by programmers false assumption that their code will run on admin level. Programs which worked without problems with user rights on Windows NT 4-5.1 won't have any problems running properly on Vista. However, common mistakes (or bad habits) like:

- Modifying files in their own Program Files directory.
- Writing in the HKEY_LOCAL_MACHINE to store settings.

Are no longer problems, since on Vista there's a thing called virtualization. Basically, every file modified in a system directory, like Program Files, is actually stored in a directory called Virtual Store. My path for this directory is C:\Users\Daniel\AppData\Local\VirtualStore. The Windows Explorer will show you those files in the system directory, but actually they are all in the VirtualStore, which, and it goes without saying, is unique for every user. The files are just stored like this in the Virtual Store:



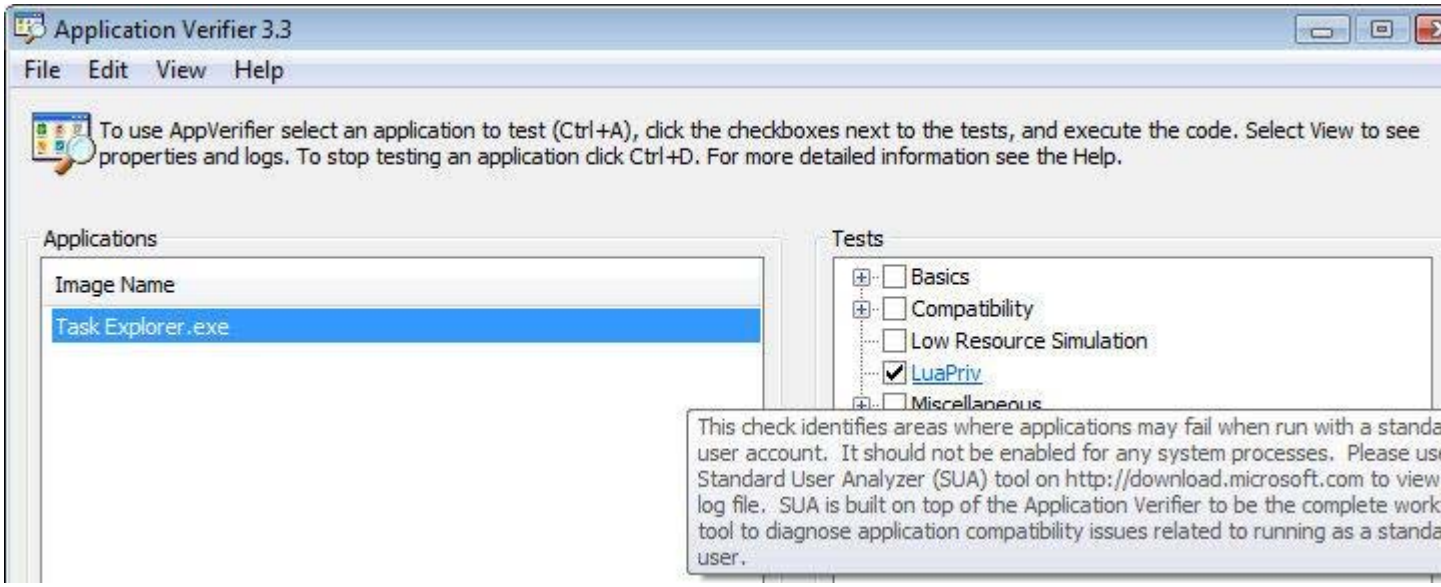
The directory hierarchy in the Virtual Store is exactly the same it would be without virtualization. I guess you have already figured out how it works.

Just like for the File System, there's also a registry virtualization. Everything written in the key *HKEY_LOCAL_MACHINE\Software* will actually be stored under *HKEY_CLASSES_ROOT\VirtualStore\Software*.

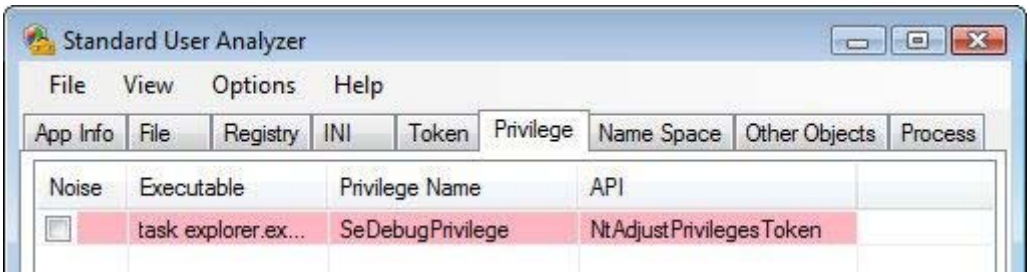
What you should keep in mind is that eveyrthing that might affect other users is no longer possible in a standard execution of your software. This means, apart from the things said above, you won't be able to load drivers, modify certain files (maybe not even read them), modify or read specific registry keys/values, access certain global objects, enumerate or modify the memory of processes which run with higher privileges than you do, etc. And you won't even be able to enumerate or send messages to windows created by those processes. This prevents exploits we have already seen in the past. There's a very good MSDN paper (Developer Best Practices and Guidelines for Applications in a Least Privileged Environment) about the UAC and its conseguences for developers, it's a detailed explanation of everything I tried to say here in short.

Compatibility Verification

There's a tool to verify if your application is compatible with the UAC. This tool is called Microsoft Application Verifier and it can also detect other issues, but right now we're not interested in those ones. Using it is very easy, all you need is to add an application to the list and select what kind of checks the verifier should do. In our case, we'll select the LUA (aka UAC) compatibility check.



The next step is running the application you added to the list. When you're done, you save the log and then, as you can read from the screenshot above, you open it with the Microsoft Standard User Analyzer. Don't even think about opening it with another application, depending on the size of your software the verifier will generate a gigantic xml log, which would make every other program consume a big part of your ram. For this test I used the Task Explorer which, of course, tries to adjust its token to SeDebugPrivilege in order to list even system processes. In fact:



Since this application is not running with admin rights, it won't be able to acquire debug privileges. I wouldn't recommend you to use always this tool, since following good design rules should be enough. However, it might be useful.

Obtaining Admin Rights

Of course, it's still possible to run an application with admin rights. You can either do that manually or through code. To do it manually, you can just right-click on the application an then click "Run as administrator" or check the "Run this program as administrator" box under Properties -> Compatibility. However, programmers cannot expect users to do these operations by themselves, so they'll need another way.

A very easy way is to tell the system the requested execution level through a manifest file. I suppose the reader knows what a manifest file is and how to integrate it in an application. There are plenty of guides about this subject and I don't think it should be re-discussed here.

The schema of such a manifest file should be like this one (this is actually for x86, as you see under processorArchitecture):

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
  <assemblyIdentity version="1.0.0.0"
    processorArchitecture="X86"
    name="YourAppName"
    type="win32"/>

 <description>Description of your application</description>
```

```
 <!-- Identify the application security requirements. -->
  <trustInfo xmlns="urn:schemas-microsoft-com:asm.v3">
    <security>
      <requestedPrivileges>
        <requestedExecutionLevel
          level="requireAdministrator"
          uiAccess="false"/>
      </requestedPrivileges>
    </security>
  </trustInfo>
</assembly>
```

The available values for **level** are:

asInvoker	The application runs with the same token as the parent process.
highestAvailable	The application runs with the highest privileges the current user can obtain.
requireAdministrator	The application runs only for administrators and requires that the application be launched with the full token of an administrator.

And for **uiAccess**:

false	The application does not need to drive input to the UI of another window on the desktop. Applications that are not providing accessibility should set this flag to false. Applications that are required to drive input to other windows on the desktop (on-screen keyboard, for example) should set this value to true.
true	The application is allowed to bypass UI protection levels to drive input to higher privilege windows on the desktop. This setting should only be used for UI Accessibility applications.

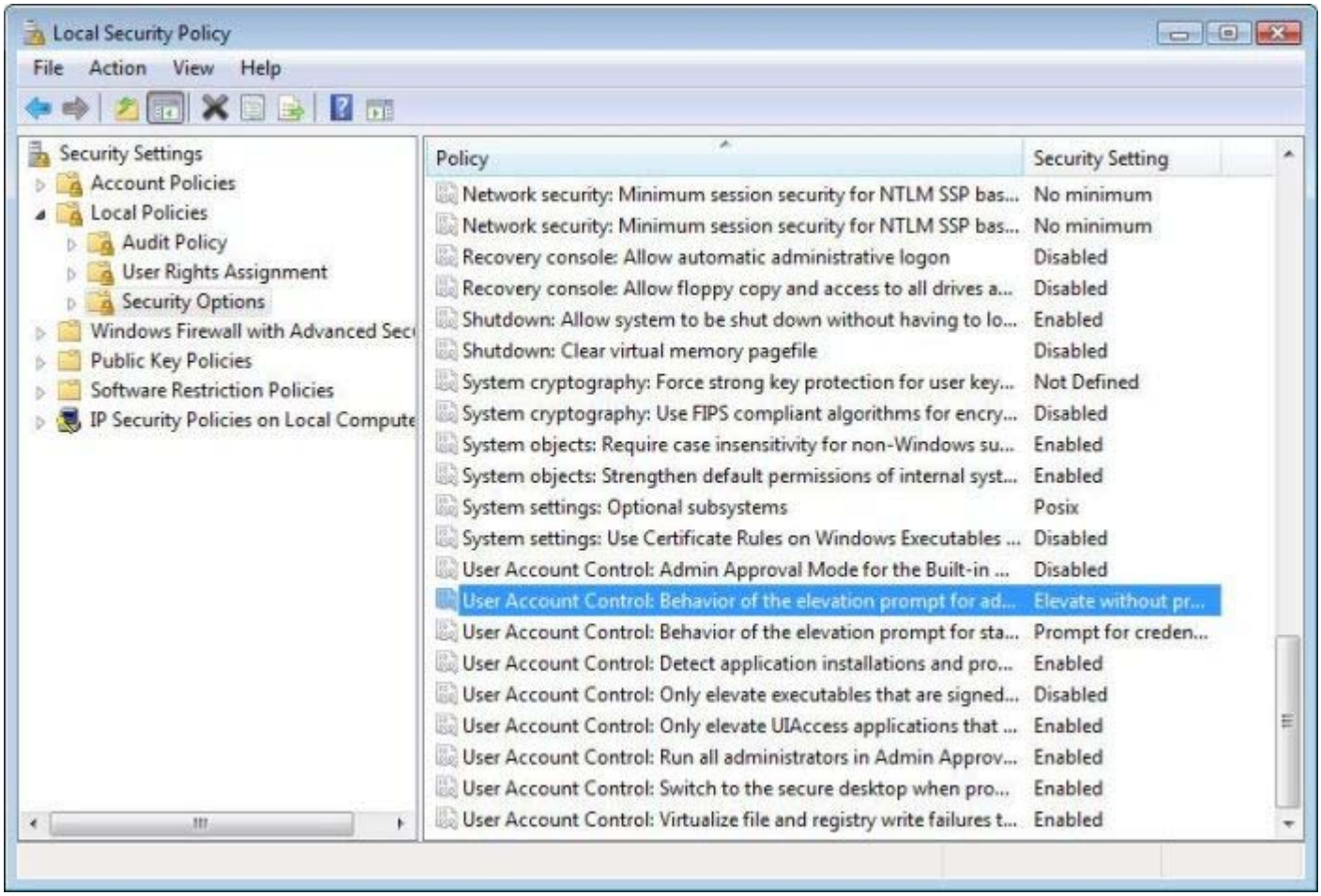
I wrote a very small application (which just waits for an input key) to demonstrate the use of a manifest file.

Open Privileges.zip (8kb) from “Files” directory inside the package.

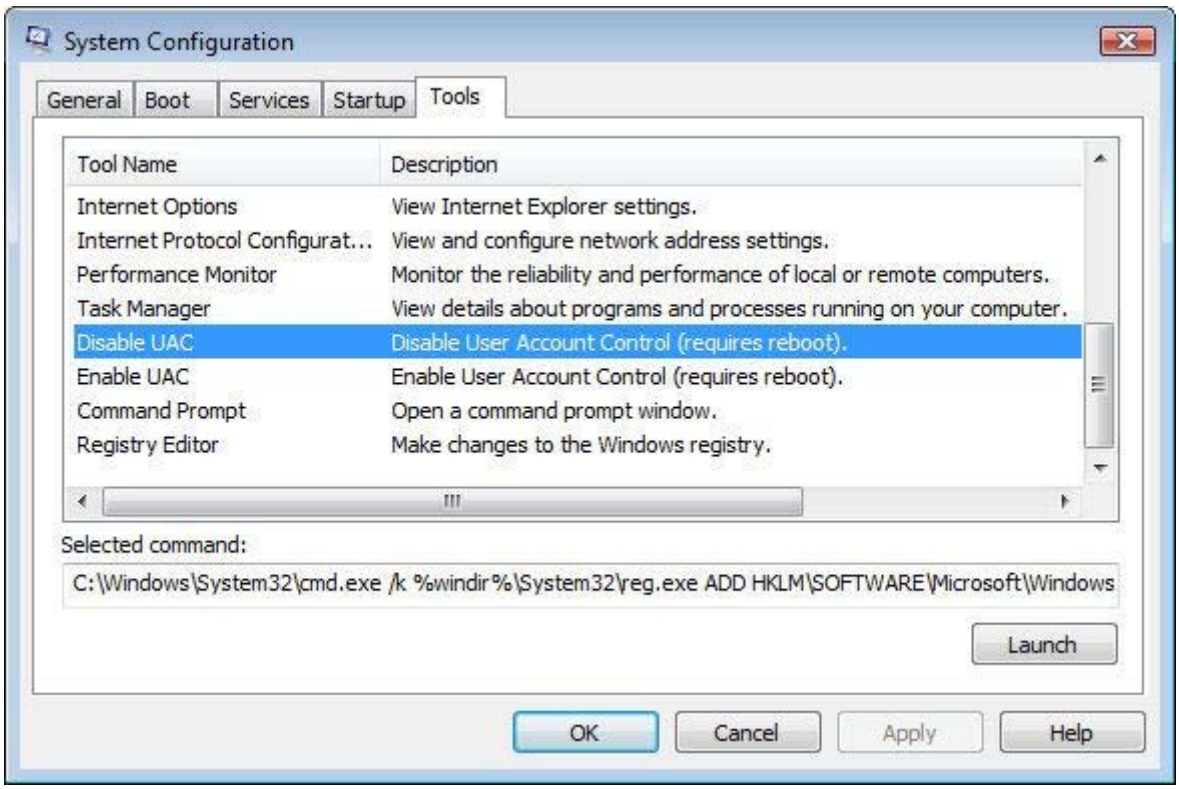
There's also a function, CredUIPromptForCredentials, to request specific credentials from a user and can be used to obtain admin rights, but it's not very comfortable for the user himself.

Disable It

If you are annoyed by all these dialogs asking you for permissions, you can disable the UAC. There are two simple ways I'm aware of. The first one described in detail on the UAC Team Blog is about going to Control Panel -> Administrative Tools -> Local Security Policy. In the tree on the left click Local Policies and then Security Options. Scroll until you find: "User Account Control: behavior of the elevation prompt for administrators". Change the current value to: "Elevate without prompting". Just like this:



The other method is more drastic. It's about removing completely the UAC and requires a machine's reboot. In Administrative Tools, instead of clicking on Local Security Policy, click on System Configuration. Choose the tab "Tools" in the dialog and scroll until you find "Disable UAC" under "Tool Name". To execute the command, click on Launch.



As you can see from the command line, it simply executes reg.exe to add a key/value. The full command line is:

C:\Windows\System32\cmd.exe /k %windir%\System32\reg.exe ADD HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System /v EnableLUA /t REG_DWORD /d 0 /f

To re-enable the UAC, it sets the EnableLUA dword to 1.

I don't know whether it's a good idea or not to disable the UAC. Sure is that a lot of programmers will be too annoyed to keep it enabled. There's been a lot of criticism about this security system. However, I personally believe it's the only one to guarantee a minimum of security on an account where one has still admin rights. If one needs more security, one shall switch to a normal user account.

Address Space Layout Randomization

In short ASLR. There's an interesting weblog about this new feature on Vista. Randomization, as you know, is useful to prevent buffer overflow exploits and has already been implemented on other operating systems. On Vista randomization is done on various levels:

- Image Base
- Stack
- Heap

With the new Service Pack 1 for Visual Studio 2005 the option /dynamicbase has been introduced. This options makes it possible to relocate executables just like dynamic libraries, just by adding a relocation section to the PE header. All executables on Windows Vista have been compiled with this option, making it impossible for a programmer to assume where specific data is placed. Randomization for images has 256 variations. The reason for this are explained by Michael Howard:

> On a final note, it is true that we don't have as much randomization as PaX and other more aggressive ASLR implementations. For instance, image randomization is only 8 bits (1 of 256 variations). Images have to be 64K aligned, and so on a 32-bit system we could have theoretically randomized images by up to 15 bits (1 of 32, 768 variations), but the incremental security gain is small - if you navigate to a Website and your browser crashes, will you go back to that site another 255 times - and would have come at the expense of fragmenting the entire address space, thereby reducing the contiguous memory available to applications and degrading system performance? We think we hit a nice balance.

I did some tests and it seems that on x86, unlike on x64, the image base changes only when rebooting. On x64 the imagebase seems changing with every execution. By the way, x64 shouldn't have problems with contiguous memory availability, since the address space is enormously larger than on x86. Unfortunately, I can't do more tests, because, at the moment, I'm running on Vista x86.

On the other hand, the stack should have 16,384 possible variations (14 bits). I wrote, for testing purposes, a little application to calculate stack variations on N executions. I wrote it very quick, so the implementation is also very rudimental. Basically, one executable calls another executable, compiled with the /dynamicbase option, N times and gets from it the stack address of a local variable. It then evaluates if the stack address has already been used or not; if not, it increments the number of variations. To communicate the stack address to the parent, the dynamic-base executable uses a SendMessage (which is enough, since wparam and lparam are the same size of a pointer). This application was compiled for both x86 and x64.

Open RandText.zip (101kb) from “Files” directory inside the package.

Here's the dynamic-base executable source:

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

#define WM_STACKADDRESS      (WM_USER + 100)
```

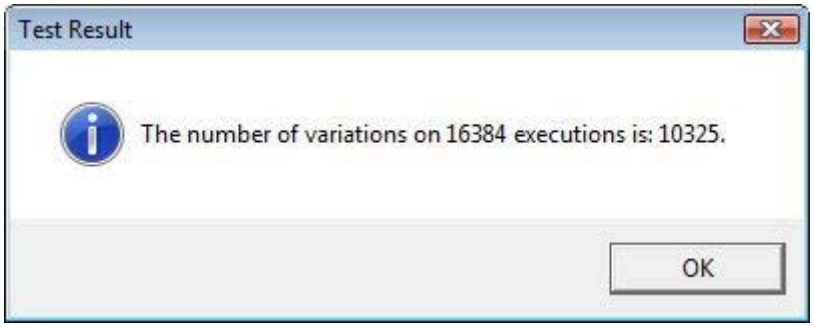
```
int APIENTRY _tWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                  LPTSTR lpCmdLine, int nCmdShow)
{
   int x;

   return (int) SendMessage(FindWindow(NULL, _T("RandTest")),
      WM_STACKADDRESS, 0, (LPARAM) &x);
}
```

And this is the main executable:

```
#include <Windows.h>
#include <tchar.h>
#include "resource.h"

#define WM_STACKADDRESS      (WM_USER + 100)

ULONG_PTR *pAddresses = NULL;
UINT nAddresses = 0;

//
// Test Loop thread: not blocking for GUI
//

void TestLoop(ULONG_PTR p)
{
   UINT nExec = (UINT) p;

   STARTUPINFO si = { 0 };
   PROCESS_INFORMATION pi = { 0 };

   for (UINT x = 0; x < nExec; x++)
   {
      CreateProcess(_T("DynBaseApp.exe"), NULL, NULL, NULL, FALSE, 0,
         NULL, NULL, &si, &pi);

      WaitForSingleObject(pi.hProcess, 3000);
   }

   delete pAddresses;

   //
   // We're through with testing, show number of variations
   //

   TCHAR szMsg[100];

   wsprintf(szMsg, _T("The number of variations on %d executions is: %d."),
      nExec, nAddresses);

   MessageBox(NULL, szMsg, _T("Test Result"), MB_ICONINFORMATION);

   ExitThread(0);
}

LRESULT CALLBACK DlgProc(HWND hDlg, UINT uMsg, WPARAM wParam, LPARAM lParam)
{


   switch (uMsg)
   {

   case WM_CLOSE:
      {
         EndDialog(hDlg, FALSE);
         break;
      }
```

```
   case WM_COMMAND:
      {
         switch ((WORD) wParam)
         {

         case IDC_TEST:
            {
               //
               // If executions != 0 start the loop
               //

               TCHAR nStrExec[20];

               GetDlgItemText(hDlg, ED_EXECUTIONS, nStrExec, 20);

               UINT nExec = _tcstoul(nStrExec, NULL, 10);

               if (nExec == 0) break;

               pAddresses = new ULONG_PTR [nExec];
               nAddresses = 0;

               CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE) &TestLoop, (LPVOID)
                  (ULONG_PTR) nExec, 0, NULL);

               break;
            }
         }

         break;
      }

   case WM_STACKADDRESS:
      {
         //
         // It's a stack address being sended by DynBaseApp.exe
         // add it to the list if not already there
         //

         ULONG_PTR Addr = (ULONG_PTR) lParam;

         for (UINT x = 0; x < nAddresses; x++)
         {
            if (Addr == pAddresses[x]) return FALSE;
         }

         pAddresses[nAddresses] = Addr;
         nAddresses++;

         break;
      }
   }

   return FALSE;
}

int APIENTRY _tWinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                  LPTSTR lpCmdLine, int nCmdShow)
{
   return (int) DialogBox(hInstance, (LPCTSTR) IDD_RANDTEST, NULL, (DLGPROC)
DlgProc);
}
```

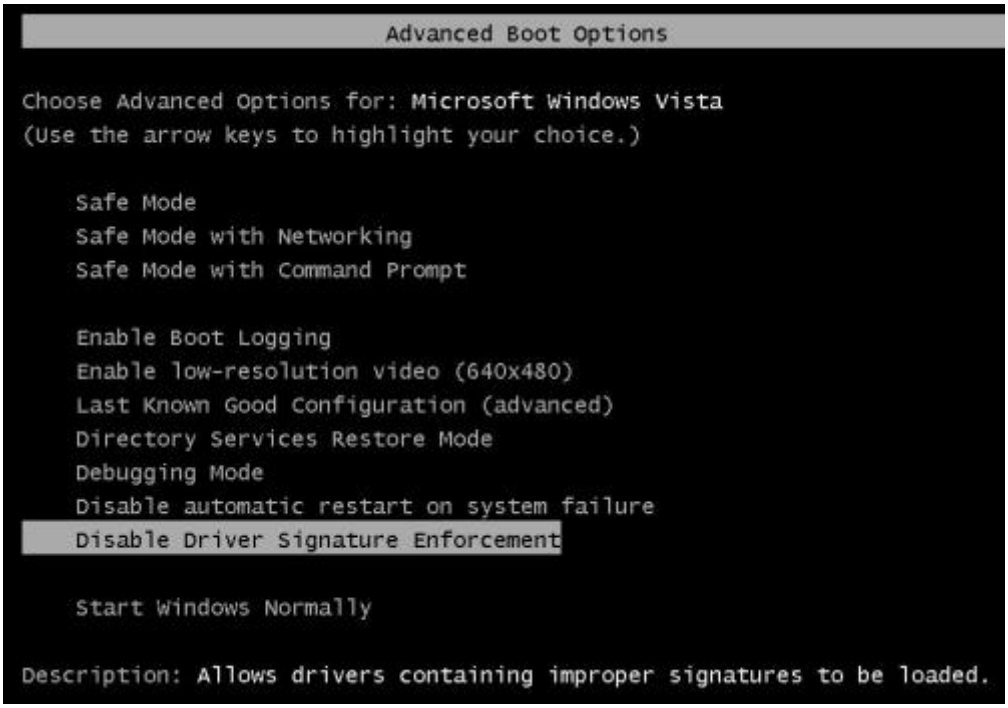
The results were pretty good. On 16,385 execution the minimum I got was 4026 (on x64). With less executions the results were, of course, even better (100/100, 199/200, 296/300).



To compile executables with the /dynamicbase option, just add "/dynamicbase" (without brackets) in the linker command line (Project -> Properties -> Linker -> Command Line -> Additional options).

Driver Signing

The big change with Vista is that drivers require now to be certificated to run (at least on 64bit), and in order to obtain the certification you have to ship an x64 version of your driver. x86-only submission are no longer accepted by WHQL (Windows Hardware Quality Labs). If you have no clue how to sign a driver there's a good doc on MSDN and, of course, the official page. Nevertheless, it's still possible to disable driver certification for testing and debugging reasons. Reboot your system and press F8 to get the Advanded Boot Options. Select "Disable Driver Signature Enforcement".



However, there's a lot more to be discussed if you're a device driver programmer or want to be one. In that case, I advise you to read the NT Insider and for this particular issue this article. It describes all the steps that have to be done to set up a Vista machine for driver debugging and testing.

Patch Guard

This paragraph (and child) shouldn't be for Vista only. However, there's been a lot of talking about Vista's Patch Guard. Patch Guard isn't news, it was first introduced for x64 (it's not available for x86) with Windows XP and 2003. I chose to put this paragraph in the Vista section, because the messing around with this technology brought some changes in Vista. What Patch Guard means is that it's no longer possible to patch kernel data, not even by trusted components. In fact, some companies, like Symantec, protested against this technology and said that Microsoft was using it in order to prevent third-part developed security solutions from working. This is non-sense, of course. Microsoft security products don't patch Vista's kernel either and use instead documented interfaces as everyone else. Basically, Patch Guard checks the integrity of system data, and if it's corrupted it

calls KeBugCheckEx causing the system to shut down. Things that trigger this behavior are:

- Modifying system service tables (SDT).
- Modifying the interrupt descriptor table (IDT).
- Modifying the global descriptor table (GDT).
- Using kernel stacks that are not allocated by the kernel.
- Patching any part of the kernel (detected only on AMD64-based systems).

Patch Guard is disabled only when debugging the system. If your product relies on patching the kernel, you might be able to use an alternative method like suggested in this weblog.

> Clearly, customers demand effective security solutions, and they can be developed without relying on kernel patching techniques. Some of the alternatives to kernel patching are:
>
> - Windows Vista includes the "Windows Filtering Platform", which enables software to perform network oriented activities such as packet inspection and other activities necessary to support firewall products.
> - The file system mini filter model allows software to participate in file system activities, which can be used by Anti-Virus software.
> - Registry notification hooks, introduced in Windows XP, and recently enhanced in Windows Vista, allow software to participate in registry related activities in the system.

I'll discuss the third alternative in the Registry Filtering paragraph.

Attacks

Some efforts have been made in this direction, since Patch Guard is a software implementation. Uninformed bypassed the Patch Guard protection on Windows XP x64. Joanna Rutkowska bypassed Vista RC1's Patch Guard by patching (on disk) the pagefile of a driver. This attack was done in user mode and therefore requires the ability to open a disk through CreateFile with write permissions and then modify data through WriteFile. It seemed that Vista's RC2 solved the issue by preventing the disk to be modified from user mode (even with high privileges). However, I've done some tests and CreateFile still returns a valid handle. The modification of some disk parts, like the boot sector, is still allowed as I could see. Nevertheless, if your utility relies on raw write access to the disk, you might encounter some problems. Fact is, this limitation doesn't even solve the issue, because theoretically the same trick could still be used in kernel mode.

Registry Filtering

Since it's no longer possible patching the Service Descriptor Table (SDT) on x64, one might be wondering how Mark Russinovich's Regmon (alias the new Process Monitor) works on x64. The answer is simple, starting with Windows XP there's no need to hook the SDT anymore. In fact, with Windows XP a new official and documented way has been introduced to filter the registry. This method relies on three functions: CmRegisterCallback (supported on XP and Vista), CmRegisterCallbackEx (supported on Vista only) and CmUnRegisterCallback. CmRegisterCallback/Ex registers a callback function for every registry operation. The callback function looks like this:

```
NTSTATUS RegistryCallback(
   IN PVOID CallbackContext,
   IN PVOID Argument1,
   IN PVOID Argument2
   );
```

These are the parameters:

CallbackCo	The value that the driver passed as the Context parameter to CmRegisterCallback or

ntext	CmRegisterCallbackEx when it registered this RegistryCallback routine.
Argument1	A REG_NOTIFY_CLASS-typed value that identifies the type of registry operation that is being performed and whether the RegistryCallback routine is being called before or after the register operation is performed.
Argument2	A pointer to a structure that contains information that is specific to the type of registry operation. The structure type depends on the REG_NOTIFY_CLASS-typed value for Argument1, as shown in the following table. For information about which REG_NOTIFY_CLASS-typed values are available for which operating system versions, see REG_NOTIFY_CLASS.

Argument1 carries the registry operation and Argument2 is a pointer to a structure. Here's the list of operations and their structures:

Operation	Structure
RegNtDeleteKey	REG_DELETE_KEY_INFORMATION
RegNtPreDeleteKey	REG_DELETE_KEY_INFORMATION
RegNtPostDeleteKey	REG_POST_OPERATION_INFORMATION
RegNtSetValueKey	REG_SET_VALUE_KEY_INFORMATION
RegNtPreSetValueKey	REG_SET_VALUE_KEY_INFORMATION
RegNtPostSetValueKey	REG_POST_OPERATION_INFORMATION
RegNtDeleteValueKey	REG_DELETE_VALUE_KEY_INFORMATION
RegNtPreDeleteValueKey	REG_DELETE_VALUE_KEY_INFORMATION
RegNtPostDeleteValueKey	REG_POST_OPERATION_INFORMATION
RegNtPostDeleteValueKey	REG_POST_OPERATION_INFORMATION
RegNtSetInformationKey	REG_SET_INFORMATION_KEY_INFORMATION
RegNtPreSetInformationKey	REG_SET_INFORMATION_KEY_INFORMATION

RegNtPostSetInformationKey	REG_POST_OPERATION_INFORMATION
RegNtRenameKey	REG_RENAME_KEY_INFORMATION
RegNtPreRenameKey	REG_RENAME_KEY_INFORMATION
RegNtPostRenameKey	REG_POST_OPERATION_INFORMATION
RegNtEnumerateKey	REG_ENUMERATE_KEY_INFORMATION
RegNtPreEnumerateKey	REG_ENUMERATE_KEY_INFORMATION
RegNtPostEnumerateKey	REG_POST_OPERATION_INFORMATION
RegNtEnumerateValueKey	REG_ENUMERATE_VALUE_KEY_INFORMATION
RegNtPreEnumerateValueKey	REG_ENUMERATE_VALUE_KEY_INFORMATION
RegNtPostEnumerateValueKey	REG_POST_OPERATION_INFORMATION
RegNtQueryKey	REG_QUERY_KEY_INFORMATION
RegNtPreQueryKey	REG_QUERY_KEY_INFORMATION
RegNtPostQueryKey	REG_POST_OPERATION_INFORMATION
RegNtQueryValueKey	REG_QUERY_VALUE_KEY_INFORMATION
RegNtPreQueryValueKey	REG_QUERY_VALUE_KEY_INFORMATION
RegNtPostQueryValueKey	REG_POST_OPERATION_INFORMATION
RegNtQueryMultipleValueKey	REG_QUERY_MULTIPLE_VALUE_KEY_INFORMATION
RegNtPreQueryMultipleValueKey	REG_QUERY_MULTIPLE_VALUE_KEY_INFORMATION

RegNtPostQueryMultipleValueKey	REG_POST_OPERATION_INFORMATION
RegNtPreCreateKey	REG_PRE_CREATE_KEY_INFORMATION
RegNtPreCreateKeyEx	REG_CREATE_KEY_INFORMATION
RegNtPostCreateKey	REG_POST_CREATE_KEY_INFORMATION
RegNtPostCreateKeyEx	REG_POST_OPERATION_INFORMATION
RegNtPreOpenKey	REG_PRE_OPEN_KEY_INFORMATION
RegNtPreOpenKeyEx	REG_OPEN_KEY_INFORMATION
RegNtPostOpenKey	REG_POST_OPEN_KEY_INFORMATION
RegNtPostOpenKeyEx	REG_POST_OPERATION_INFORMATION
RegNtKeyHandleClose	REG_KEY_HANDLE_CLOSE_INFORMATION
RegNtPreKeyHandleClose	REG_KEY_HANDLE_CLOSE_INFORMATION
RegNtPostKeyHandleClose	REG_POST_OPERATION_INFORMATION
RegNtPreFlushKey	REG_FLUSH_KEY_INFORMATION
RegNtPostFlushKey	REG_POST_OPERATION_INFORMATION
RegNtPreLoadKey	REG_LOAD_KEY_INFORMATION
RegNtPostLoadKey	REG_POST_OPERATION_INFORMATION
RegNtPreUnLoadKey	REG_UNLOAD_KEY_INFORMATION
RegNtPostUnLoadKey	REG_POST_OPERATION_INFORMATION
RegNtPreQueryKeySec	REG_QUERY_KEY_SECURITY_INFOR

urity	MATION
RegNtPostQueryKeySecurity	REG_POST_OPERATION_INFORMATION
RegNtPreSetKeySecurity	REG_SET_KEY_SECURITY_INFORMATION
RegNtPostSetKeySecurity	REG_POST_OPERATION_INFORMATION
RegNtCallbackContextCleanup	REG_CALLBACK_CONTEXT_CLEANUP_INFORMATION

According to the MSDN, pointers in these structures should be accessed in try/except blocks. The callback can prevent operations from being performed as well (it's a real filter). To do that on XP, it just has to return a value different from STATUS_SUCCESS. Unfortunately, by doing this, the thread which originally called the registry function will get the same error as well. That's why on Vista a new value is supported: STATUS_CALLBACK_BYPASS. By returning this value, the registry operation isn't actually performed, but the thread won't get an error value. This is very useful for security solutions.

I wrote a small (very small) registry filter to show how this new method works. Don't get too excited about it, I wrote it in 20 minutes and it's not that good, but maybe it's helpful for someone.

Open MyRegFilter.zip (9kb) from “Files” directory inside the package.

```
#include <ntddk.h>

WCHAR DeviceName[] = L"\\Device\\MyRegFilter";
WCHAR SymLinkName[] = L"\\DosDevices\\MyRegFilter";

UNICODE_STRING usDeviceName;
UNICODE_STRING usSymbolicLinkName;

typedef struct _DEVICE_CONTEXT
{
   PDRIVER_OBJECT  pDriverObject;
   PDEVICE_OBJECT  pDeviceObject;

   LARGE_INTEGER RegCookie;
}
DEVICE_CONTEXT, *PDEVICE_CONTEXT, **PPDEVICE_CONTEXT;

PDEVICE_OBJECT  g_pDeviceObject  = NULL;
PDEVICE_CONTEXT g_pDeviceContext = NULL;

#define FILE_DEVICE_MYREGFILTER 0x8000


NTSTATUS DriverInitialize(PDRIVER_OBJECT  pDriverObject, PUNICODE_STRING
pusRegistryPath);
NTSTATUS DriverEntry(PDRIVER_OBJECT  pDriverObject, PUNICODE_STRING pusRegistryPath);

NTSTATUS RegistryCallback(PVOID CallbackContext, PVOID Argument1, PVOID Argument2);

#ifdef ALLOC_PRAGMA

#pragma alloc_text (INIT, DriverInitialize)
#pragma alloc_text (INIT, DriverEntry)

#endif
```

```
NTSTATUS DeviceDispatcher(PDEVICE_CONTEXT pDeviceContext, PIRP pIrp)
{
    PIO_STACK_LOCATION pisl;
    NTSTATUS ns = STATUS_NOT_IMPLEMENTED;

    pisl = IoGetCurrentIrpStackLocation(pIrp);

    switch (pisl->MajorFunction)
    {

    case IRP_MJ_CREATE:
    case IRP_MJ_CLEANUP:
    case IRP_MJ_CLOSE:
    case IRP_MJ_DEVICE_CONTROL:
       {
            ns = STATUS_SUCCESS;
            break;
       }
    }

    pIrp->IoStatus.Status = ns;
    pIrp->IoStatus.Information = 0;

    IoCompleteRequest(pIrp, IO_NO_INCREMENT);

    return ns;
}

NTSTATUS DriverDispatcher(PDEVICE_OBJECT pDeviceObject, PIRP pIrp)
{
   return (pDeviceObject == g_pDeviceObject ?
      DeviceDispatcher(g_pDeviceContext, pIrp) : STATUS_INVALID_PARAMETER_1);
}


VOID DriverUnload(PDRIVER_OBJECT pDriverObject)
{
   //
   // Stop filtering the registry
   // Shouldn't be placed in the unload
   //

   CmUnRegisterCallback(g_pDeviceContext->RegCookie);

   IoDeleteSymbolicLink(&usSymbolicLinkName);
   IoDeleteDevice(pDriverObject->DeviceObject);
}

NTSTATUS DriverInitialize(PDRIVER_OBJECT pDriverObject,
                          PUNICODE_STRING pusRegistryPath)
{
    PDEVICE_OBJECT pDeviceObject = NULL;
    NTSTATUS ns = STATUS_DEVICE_CONFIGURATION_ERROR;

    RtlInitUnicodeString(&usDeviceName, DeviceName);
    RtlInitUnicodeString(&usSymbolicLinkName, SymLinkName);

    if ((ns = IoCreateDevice(pDriverObject, sizeof (DEVICE_CONTEXT),
       &usDeviceName, FILE_DEVICE_MYREGFILTER, 0, FALSE,
       &pDeviceObject)) == STATUS_SUCCESS)
    {
        if ((ns = IoCreateSymbolicLink(&usSymbolicLinkName,
         &usDeviceName)) == STATUS_SUCCESS)
      {
            g_pDeviceObject  = pDeviceObject;
            g_pDeviceContext = pDeviceObject->DeviceExtension;
```

```
            g_pDeviceContext->pDriverObject = pDriverObject;
            g_pDeviceContext->pDeviceObject = pDeviceObject;

      }
      else
      {
         IoDeleteDevice(pDeviceObject);
      }
    }

   return ns;
}

NTSTATUS DriverEntry(PDRIVER_OBJECT pDriverObject, PUNICODE_STRING pusRegistryPath)
{
   PDRIVER_DISPATCH *ppdd;
   NTSTATUS ns = STATUS_DEVICE_CONFIGURATION_ERROR;

   if ((ns = DriverInitialize(pDriverObject, pusRegistryPath)) == STATUS_SUCCESS)
   {
        ppdd = pDriverObject->MajorFunction;

        ppdd[IRP_MJ_CREATE                   ] =
        ppdd[IRP_MJ_CREATE_NAMED_PIPE        ] =
        ppdd[IRP_MJ_CLOSE                    ] =
        ppdd[IRP_MJ_READ                     ] =
        ppdd[IRP_MJ_WRITE                    ] =
        ppdd[IRP_MJ_QUERY_INFORMATION        ] =
        ppdd[IRP_MJ_SET_INFORMATION          ] =
        ppdd[IRP_MJ_QUERY_EA                 ] =
        ppdd[IRP_MJ_SET_EA                   ] =
        ppdd[IRP_MJ_FLUSH_BUFFERS            ] =
        ppdd[IRP_MJ_QUERY_VOLUME_INFORMATION] =
        ppdd[IRP_MJ_SET_VOLUME_INFORMATION   ] =
        ppdd[IRP_MJ_DIRECTORY_CONTROL        ] =
        ppdd[IRP_MJ_FILE_SYSTEM_CONTROL      ] =
        ppdd[IRP_MJ_DEVICE_CONTROL           ] =
        ppdd[IRP_MJ_INTERNAL_DEVICE_CONTROL  ] =
        ppdd[IRP_MJ_SHUTDOWN                 ] =
        ppdd[IRP_MJ_LOCK_CONTROL             ] =
        ppdd[IRP_MJ_CLEANUP                  ] =
        ppdd[IRP_MJ_CREATE_MAILSLOT          ] =
        ppdd[IRP_MJ_QUERY_SECURITY           ] =
        ppdd[IRP_MJ_SET_SECURITY             ] =
        ppdd[IRP_MJ_POWER                    ] =
        ppdd[IRP_MJ_SYSTEM_CONTROL           ] =
        ppdd[IRP_MJ_DEVICE_CHANGE            ] =
        ppdd[IRP_MJ_QUERY_QUOTA              ] =
        ppdd[IRP_MJ_SET_QUOTA                ] =
        ppdd[IRP_MJ_PNP                      ] = DriverDispatcher;
        pDriverObject->DriverUnload            = DriverUnload;

      //
      // Filter the registry
      //

      ns = CmRegisterCallback(RegistryCallback, g_pDeviceContext, &g_pDeviceContext-
>RegCookie);

      if (!NT_SUCCESS(ns)) IoDeleteDevice(g_pDeviceObject);
   }

   return ns;
}

//
// Registry Filter Callback
//
```

```
NTSTATUS RegistryCallback(PVOID CallbackContext, PVOID Argument1, PVOID Argument2)
{
   PDEVICE_CONTEXT pContext = (PDEVICE_CONTEXT) CallbackContext;
   REG_NOTIFY_CLASS Action  = (REG_NOTIFY_CLASS) Argument1;


   switch (Action)
   {

   case RegNtPreDeleteKey:
      {
         //
         // Pre DeleteKey
         //

         PREG_DELETE_KEY_INFORMATION pInfo = (PREG_DELETE_KEY_INFORMATION) Argument2;

         DbgPrint("Delete Key\n");

         //
         // You can prevent this operation form happening
         // Without having the thread noticing it
         // Only on Windows Vista
         //
         //
         // return STATUS_CALLBACK_BYPASS;
         //

         break;
      }

   case RegNtPreCreateKeyEx:
      {
         //
         // Pre CreateKey
         //

         PREG_CREATE_KEY_INFORMATION pInfo = (PREG_CREATE_KEY_INFORMATION) Argument2;

         DbgPrint("Create Key\n");

         break;
      }


   default:
      {
         //
         // Return STATUS_SUCCESS
         //

         break;
      }

   }

   return STATUS_SUCCESS;
}
```

In this code sample I use DbgPrint to be notified of registry operations. On Vista the output of DbgPrint is disabled by default. If you want to enable it, follow these instructions.

Power Management

Power Management was improved in Vista, not only because new things have been introduced, but also because the Power Management for device driver programmers has been made easier. These

WinHec docs are a very good source to start from. One of the big news is the introduction of the Hybrid Sleep (the default off-mode). In this sleep mode the system image is written on a hybernation file on disk from where the system can be resumed. Drivers are notified of entering the Hybrid Sleep (S4 state) by IRP_MN_SET_POWER (Parameters.Power.State == PowerSystemHybernate). Use the SYSTEM_POWER_STATE_CONTEXT structure (Parameters.Power.SystemPowerStateContext) to determine the state transition process.

Also, it might not be very important, but I happened to read what follows. Silent shutdown cancellations (in user mode) are no longer allowed on Vista. This means that if your application gets a WM_QUERYENDSESSION and doesn't return TRUE (in order to let the system shut down), Vista will show a dialog box informing the user of this behavior.

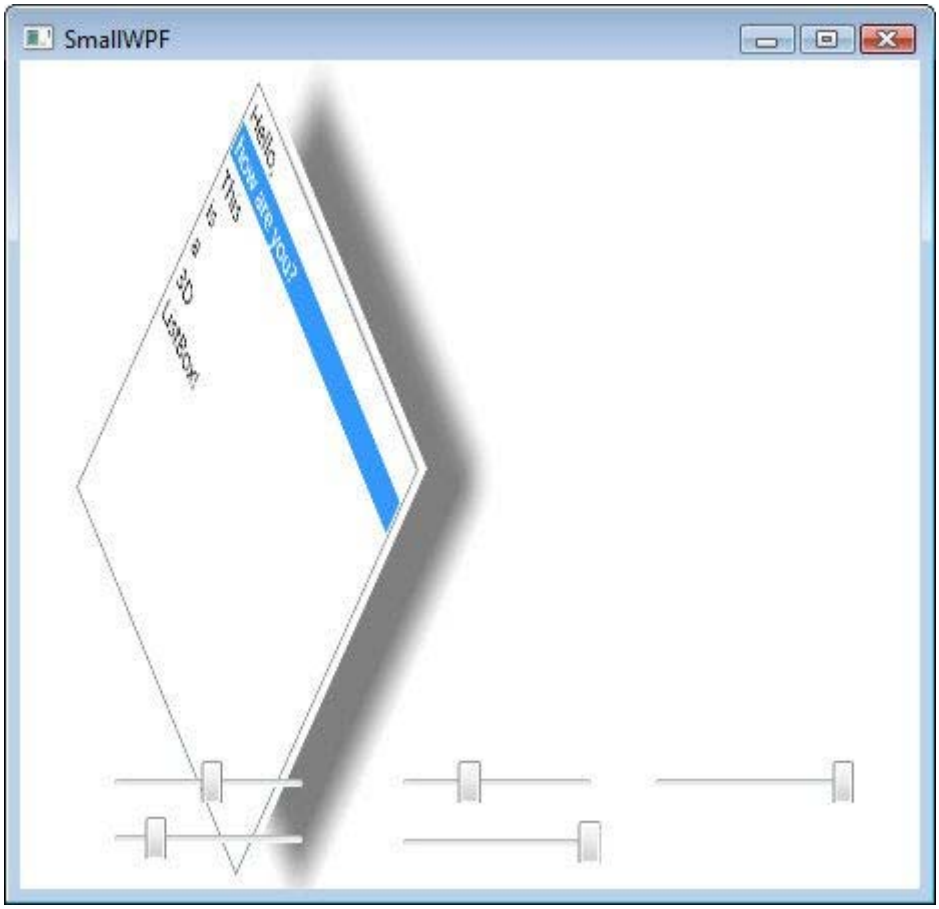
.NET Framework 3.0

The .NET Framework 3.0 is shipped along with Vista. However, it can be installed on XP SP2 as well. To use the new technologies introduced by this new framework with your Visual Studio 2005, you'll need two extensions. One for the Windows Presentation Foundation (WPF) and Windows Communication Foundation (WCF). And one for the Windows Workflow Foundation (WWF). I cannot discuss these technologies extensively, of course, but I can try to give an insight to programmers who have never worked with them.

Windows Presentation Foundation

I'm very enthusiastic about this technology, but it takes a moment or two for old fashioned C/C++ programmers (like myself) to understand how it works. Basically, it's a new way of creating GUIs for desktop applications and web pages. The main difference from the old way, is that these GUIs are created through XAML (eXtensible Application Markup Language), a language based on XML. The advantages of using the WPF are many. You can use 2D/3D, audio, video, animations etc. in seconds. There are no more HWNDs, and all the work is delegated to the GPU. On MSDN TV there are a few demonstrations of how through the WPF you can design beautiful and advanced GUIs. The WPF offers a very good separation between GUI development and the internal code implementation. Also, a lot of things can be achieved through XAML without having to use C#/VB code.

Open SmallWPF.zip (374kb) from “Files” directory inside the package.



In this little code sample I bind sliders to values of a listbox in order to change its appearance (position and shadow). What's so stunning is that the listbox can still be used, you can scroll it, select items, etc. I don't want to say that rotating a listbox is useful, but this is just a sample of what can be done. As I said, the slider are bound to values, this means that I didn't use code. Everything this application does is written in XAML. Here's all the code:

```
<Window x:Class="SmallWPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="SmallWPF" Height="339" Width="454" xmlns:my="clr-
namespace:System;assembly=mscorlib">
  <Grid>
    <Border BorderBrush="White" BorderThickness="5" HorizontalAlignment="Center"
      VerticalAlignment="Top">
      <ListBox Width="200" Height="200" Name="listBox1" >
        <ListBoxItem>Hello,</ListBoxItem>
        <ListBoxItem IsSelected="True">how are you?</ListBoxItem>
        <ListBoxItem>This</ListBoxItem>
        <ListBoxItem>is</ListBoxItem>
        <ListBoxItem>a</ListBoxItem>
        <ListBoxItem>3D</ListBoxItem>
        <ListBoxItem>ListBox!</ListBoxItem>
      </ListBox>
      <Border.BitmapEffect>
        <BitmapEffectGroup>
          <DropShadowBitmapEffect Color="Black"
            Direction="{Binding ElementName=MySlider4, Path=Value}"
            ShadowDepth="{Binding ElementName=MySlider5, Path=Value}"
            Softness="1" Opacity="0.5"/>
        </BitmapEffectGroup>
      </Border.BitmapEffect>
      <Border.RenderTransform>
        <TransformGroup>
          <SkewTransform CenterX="0" CenterY="0"
            AngleX="{Binding ElementName=MySlider1, Path=Value}"
            AngleY="{Binding ElementName=MySlider2, Path=Value}" />
          <RotateTransform Angle="{Binding ElementName=MySlider3, Path=Value}" />
```

```
        </TransformGroup>
      </Border.RenderTransform>
    </Border>
    <Slider Height="21" Margin="42,0,0,43" Name="MySlider1"
VerticalAlignment="Bottom"
      HorizontalAlignment="Left" Width="104" Minimum="0" Maximum="50" />
    <Slider Height="21" Margin="184,0,158,43" Name="MySlider2"
VerticalAlignment="Bottom"
      Width="104" Minimum="0" Maximum="50" />
    <Slider Height="21" Margin="0,0,33,43" Name="MySlider3"
VerticalAlignment="Bottom"
      HorizontalAlignment="Right" Width="104" Minimum="0" Maximum="50" />
    <Slider Height="21" Margin="42,0,0,15" Name="MySlider4"
VerticalAlignment="Bottom"
      Width="104" Minimum="0" Maximum="200" HorizontalAlignment="Left" />
    <Slider Height="21" Margin="184,0,158,13" Name="MySlider5"
VerticalAlignment="Bottom"
      Width="104" Minimum="0" Maximum="100" />
  </Grid>
</Window>
```

I used this code to bind a value to a slider:

```
AngleX="{Binding ElementName=MySlider1, Path=Value}"
```

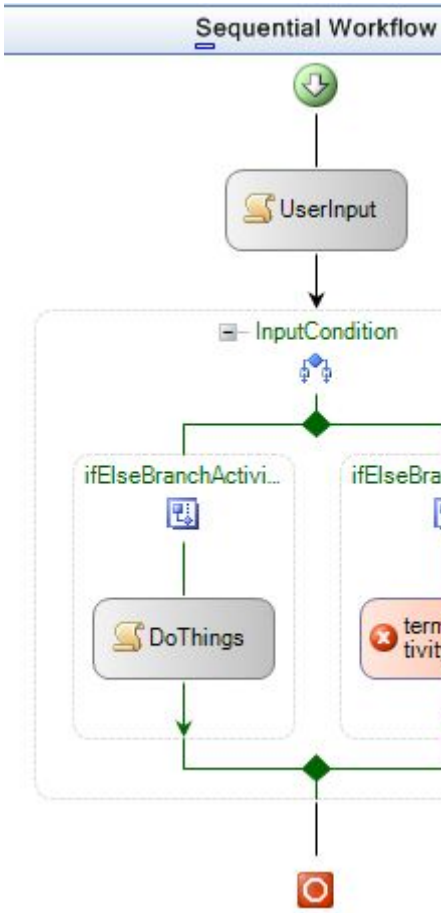
ElementName is the name of the control to bind and Path is the property of the bound control which should be used to fill the value. In this case, the position of MySlider1 fills the AngleX field. I could also bind a control's behavior to C# code. But, of course, this is not the place to discuss every property of this technology. I just hope that this paragraph got you intrigued enough to make you want to read more about it.

Windows Communication Foundation

The WCF is an interface to design services. The background idea is unified programming model for already existing technologies: COM+ / .NET Enterprise services, MSMQ, .NET Remoting, ASP.NET Web Services, Web Services Enhancements (WSE). Moreover, the ability of intercommunication between these technologies handled by the WCF, without having the programmer to worry about it, through reliable and secure ways. From what I could read, it seems a very good final solution to all the problems we know from the past, since from now on the programmer doesn't have to think about the communication process itself, which is handled by the WCF, meaning he doesn't have to worry about which technology he is communicating with and where from. For more information, this is the official Windows Communication Foundation hompage. However, there are even more practical samples on codeproject.

Windows Workflow Foundation

The WWF is a good way of formalizing workflow-based activities through visual items which are bound to code. Ok, this sounds strange, I'll try again. Basically, if you are a company and have to formalize a process of activities and want the comfort of having a visual model, then the WWF is what you are looking for. I can't talk about this subject extensively because I haven't used this technology extensively myself. However, since many programmers might wonder what this technology is all about, I'll try a simple understanding approach. Here's a workflow graph I made:



As you can see, the graph is divided in single activities (very few, because I have no imagination). The activities can be bound through declarative rules. This means that if a have a condition that needs to be satisfied, I can declare the condition as a property. There are a many workflow components, each of them has its own properties. For instance, the code-workflow component can define a code function in a C# file, which is executed when it's the component turn of activity. In this little sample, the first activity is to wait for the user's input. After that, a declarative rule is set, which sub-divides the workflow in two separate activity flows.

I hope that, in spite of my terrible workflow model, you got the general purpose of the Windows Workflow Foundation. If you're interested in learning more, check out the official homepage where you can also find a lot of code samples. As usual, there's plenty of guides about this subject.

Conclusions

It's over now. I hope you enjoyed the article and didn't dislike the idea of such a general overview about two really extensive subjects like x64 and Windows Vista. I noticed during the writing that I had to put a lot of images in the article and that this might be problem for slow connections. I'm sorry for that, but this is the direct consequence of not subdividing this paper in more articles.

Ntoskrnl