Article • 10/07/2019 • 29 minutes to read

New information has been added to this article since publication.
Refer to the Editor's Update below.

x64 Primer

Everything You Need To Know To Start Programming 64-Bit Windows Systems

Matt Pietrek

This article discusses:

- Background on 64-bit versions of Windows
- Just enough x64 architecture to get by
- Developing for x64 with Visual C++ 2005
- Debugging techniques for your x64 builds

This article uses the following technologies:
Windows, Win64, Visual Studio 2005

Contents

One of the pleasures of working on the bleeding edge of Windows® is poking around in a new technology to see how it works. I don't really feel comfortable with an operating system until I have a little under-the-hood knowledge. So when the 64-bit editions of Windows XP and Windows Server™ 2003 appeared on the scene, I was all over them.

The nice thing about Win64 and the x64 CPU architecture is that they're different enough from their predecessors to be interesting, while not requiring a huge learning curve. While we developers would like to think that moving to x64 is just a recompile away, the reality is

that we'll still spend far too much time in the debugger. A good working knowledge of the OS and CPU is invaluable.

In this article I'll boil down my experiences with Win64 and the x64 architecture to the essentials that a hotshot Win32® programmer needs for the move to x64. I'll assume that you know basic Win32 concepts, basic x86 concepts, and why your code should run on Win64. This frees me to focus on the good stuff. Think of this overview as a look at just the important differences relative to your knowledge of Win32 and the x86 architecture.

One nice thing about x64 systems is that you can use either Win32 or Win64 on the same machine without serious performance losses, unlike Itanium-based systems. And despite a few obscure differences between the Intel and AMD x64 implementations, the same x64-compatible build of Windows should run on either. You don't need one version of Windows for AMD x64 systems and another for Intel x64 systems.

I've divided the discussion into three broad areas: OS implementation details, just enough x64 CPU architecture to get by, and developing for x64 with Visual C++®.

The x64 Operating System

In any overview of the Windows architecture, I like to start with memory and the address space. Although a 64-bit processor could theoretically address 16 exabytes of memory (2^64), Win64 currently supports 16 terabytes, which is represented by 44 bits. Why can't you just load a machine up with 16 exabytes to use all 64 bits? There are a number of reasons.

For starters, current x64 CPUs typically only allow 40 bits (1 terabyte) of physical memory to be accessed. The architecture (but no current hardware) can extend this to up to 52 bits (4 petabytes) Even if that restriction was removed, the size of the page tables to map that much memory would be enormous.

Just as in Win32, the addressable range is divided into user and kernel mode areas. Each process gets its own unique 8TB at the bottom end, while kernel mode code lives in the upper 8 terabytes and is shared by all processes. The different versions of 64-bit Windows have different physical memory limits as shown in **Figure 1** and **Figure 2**.

Figure 2 Physical Memory and CPU Limits

Physical Memory and CPU Limits	32-Bit Models	64-Bit Models
Windows XP Professional	4GB (1-2 CPUs)	128GB (1-2 CPUs)

Physical Memory and CPU Limits	32-Bit Models	64-Bit Models
Windows Server 2003, Standard Edition	4GB (1-4 CPUs)	32GB (1-4 CPUs)
Windows Server 2003, Enterprise Edition	64GB (1-8 CPUs)	1TB (1-8 CPUs)
Windows Server 2003, Datacenter Edition	64GB (8-32 CPUs)	1TB (8-64 CPUs)

Figure 1 General Memory Limits

	32-Bit Models	64-Bit Models
Total virtual address space (based on a single process)	4GB	16TB
Virtual address space per 32-bit process	2GB (3GB if system is booted with /3GB switch)	4GB if compiled with /LARGEADDRESSAWARE (2GB otherwise)
Virtual address space per 64-bit process	Not applicable	8TB
Paged pool	470MB	128GB
Non-paged pool	256MB	128GB
System Page Table Entry (PTE)	660MB to 900MB	128GB

Also just like in Win32, the x64 page size is 4KB. The first 64KB of address space is never mapped in, so the lowest valid address you'd expect to see is 0x10000. Unlike in Win32, system DLLs don't have a default load address near the top of the user mode address range. Instead, they're loaded above 4GB, typically at addresses around 0x7FF00000000.

A nice feature of many newer x64 processors is support for the CPU No Execute bit that Windows uses to implement hardware Data Execution Protection (DEP). On the x86 platform, many bugs and viruses exist because the CPU can execute data as if it were legal code bytes. A buffer overrun (intentional or not) can end up with the CPU executing through memory that was intended for data storage. With DEP, the OS can set much more clear boundaries around valid code regions, thus causing the CPU to trap if execution goes outside these expected boundaries. This helps in the continuing battle to make Windows less vulnerable to attack.

In a move designed to catch errors, the x64 linker assigns default load addresses for executables just above 32 bits (4GB). This helps you to quickly find these areas in existing

code after the code has been ported to Win64. Specifically, if a pointer is stored in a 32-bit sized value (a DWORD, for example), it will effectively be truncated when running in a Win64 build, making the pointer invalid and thus triggering an access violation. This trick makes it much easier to find those nasty pointer bugs.

[**Editor's Update - 5/2/2006:** Handles are defined as pointer values. Thus in Win64, a handle is 8 bytes, not 4 bytes.]

The file format for Win64 is called PE32+. From nearly every viewpoint, the format is structurally identical to the Win32 PE file. A very few fields such as the ImageBase in the header have been widened, one field was deleted, and one field was changed to reflect a different CPU type. **Figure 3** shows the fields that have changed.

Figure 3 Changes to PE File FIelds

Header Field	Change
Magic	Set to 0x20b instead of 0x10b
BaseOfData	Deleted
ImageBase	Widened to 64 bits
SizeOfStackReserve	Widened
SizeOfStackCommit	Widened
SizeOfHeapReserve	Widened
SizeOfHeapCommit	Widened

Beyond the PE header, there aren't many changes. A few structures such as IMAGE_LOAD_CONFIG and IMAGE_THUNK_DATA simply had some of their fields widened to 64 bits. The addition of the PDATA section is more interesting, as it highlights one of the major differences between the Win32 and Win64 implementation: exception handling.

In the x86 world, exception handling is stack-based. When a Win32 function contains try/catch or try/finally code, the compiler emits instructions that create a small data block on the stack. In addition, each try data block points to the previous try data structure, thus forming a linked list with the most recently added structure at the list head. As functions are called and exited, the head of the linked list keeps updating. When an exception occurs, the OS walks the linked list of blocks on the stack, looking for the appropriate handler. My

January 1997 *MSJ* article describes this in much more detail, so I'll keep the description to a minimum here.

In contrast to the Win32 exception handling, Win64 (both x64 and Itanium versions) uses table-based exception handling. No linked list of try data blocks is built on the stack. Instead, each Win64 executable contains a runtime function table. Each function table entry contains both the starting and ending address for the function, as well as the location of a rich set of data about exception-handling code in the function and the function's stack frame layout. See the IMAGE_RUNTIME_FUNCTION_ENTRY structure in WINNT.H and in the x64 SDK for the nitty-gritty on these structures.

When an exception occurs, the OS walks the regular thread stack. As the stack walk encounters each frame and saved instruction pointer, the OS determines within which executable module the instruction pointer lies. The OS then searches the runtime function table in that module, locates the appropriate runtime function entry, and makes the appropriate exception-processing decisions from that data.

What if you're a rocket scientist and you've generated code directly in memory without an underlying PE32+ module? You're still covered in this case. Win64 has a RtlAddFunctionTable API that lets you tell the OS about your dynamically generated code.

The downside to table-based exception handling (relative to the x86 stack-based model) is that looking up function table entries from code addresses takes more time than just walking a linked list. The upside is that functions don't have the overhead of setting up a try data block every time the function executes.

Remember, this is just a quick introduction rather than a full fledged description of x64 exception processing, however exciting that might be! For a more in-depth overview of the x64 exception model, be sure to read Kevin Frei's blog entry .

x64-compatible versions of Windows don't contain dramatic numbers of truly new APIs; most new Win64 APIs were added to the Windows releases for Itanium processors. In the interest of keeping things brief, the main two existing APIs of importance are IsWow64Process and GetNativeSystemInfo. These allow Win32 apps to determine if they're running on Win64, and if so, the true capabilities of the system. Otherwise, a 32-bit process that calls GetSystemInfo only sees the system capabilities as if it's a 32-bit system. For instance, GetSystemInfo can only report the address range of 32-bit processes. **Figure 4** shows the APIs that were not previously available on x86, but are available for x64.

Figure 4 New 64-Bit APIs

Functionality	API
Exception Handling	RtlAddFunctionTable RtlDeleteFunctionTable RtlRestoreContext RtlLookupFunctionEntry RtlInstallFunctionTableCallback
Registry	RegDeleteKeyEx RegGetValue RegQueryReflectionKey
NUMA (Non-Uniform Memory Access)	GetNumaAvailableMemoryNode GetNumaHighestNodeNumber GetNumaNodeProcessorMask GetNumaProcessorNode
WOW64 Redirection	Wow64DisableWow64FsRedirection Wow64RevertWow64FsRedirection RegDisableReflectionKey RegEnableReflectionKey
Miscellaneous	GetLogicalProcessorInformation QueryWorkingSetEx SetThreadStackGuarantee GetSystemFileCacheSize SetSystemFileCacheSize EnumSystemFirmwareTables GetSystemFirmwareTable

While running a fully 64-bit Windows system sounds great, the reality is that you'll very likely need to run Win32 code for a while. Towards that end, x64 versions of Windows include the WOW64 subsystem that lets Win32 and Win64 processes run side-by-side on the same system. However, loading your 32-bit DLL into a 64-bit process, or vice versa, isn't supported. (It's a good thing, trust me.) And you can finally kiss good bye to 16-bit legacy code!

In x64 versions of Windows, a process that starts from a 64-bit executable such as Explorer.exe can only load Win64 DLLs, while a process started from a 32-bit executable can only load Win32 DLLs. When a Win32 process makes a call to kernel mode—to read a file, for instance—the WOW64 code intercepts the call silently and invokes the correct x64 equivalent code in its place.

Of course, processes of different lineages (32-bit versus 64-bit) need to be able to communicate with each other. Luckily, all the usual interprocess communication mechanisms that you know and love in Win32 also work in Win64, including shared memory, named pipes, and named synchronization objects.

You might be thinking, "But what about the system directory? The same directory can't hold both 32- and 64-bit versions of system DLLs such as KERNEL32 or USER32, right?" WOW64 magically takes care of this for you by the doing selective file system redirection. File activity from a Win32 process that would normally go to the System32 directory instead goes to a directory named SysWow64. Under the covers, WOW64 silently changes these requests to point at the SysWow64 directory. A Win64 system effectively has two

\Windows\System32 directories—one with x64 binaries, the other with the Win32 equivalents.

Smooth as it may seem, this can be confusing. For instance, I was at one point using (unbeknownst to me) a 32-bit command-line prompt. When I ran DIR on Kernel32.dll in the System32 directory, I got the exact same results as when I did the same thing in the SysWow64 directory. It took a lot of head scratching before I figured out that the file system redirection was working just like it should. That is, even though I thought I was working in the \Windows\System32 directory, WOW64 was redirecting the calls to the SysWow64 directory. Incidentally, if you really do want to get at the 32-bit \Windows\System32 directory from an x64 app, the GetSystemWow64Directory API gives you the correct path. Be sure to read the MSDN® documentation for the complete story.

Beyond file system redirection, another bit of magic performed by WOW64 is registry redirection. Consider my earlier statement about Win32 DLLs not loading into Win64 processes, and then think about COM and its use of the registry to load in-process server DLLs. What if a 64-bit application uses CoCreateInstance to create an object that's implemented in a Win32 DLL? The DLL can't load, right? WOW64 saves the day again by redirecting access from 32-bit applications to the \Software\Classes (and related) registry nodes. The net effect is that Win32 applications have a different (but mostly parallel) view of the registry than x64 applications. As you'd expect, the OS provides an escape hatch for 32-bit applications to read the actual 64-bit registry value by specifying new flag values when calling RegOpenKey and friends.

Drilling down a bit, the last few OS differences near and dear to my heart concern thread local data. In x86 versions of Windows, the FS register is used to point at per-thread memory areas, including the "last error" and Thread Local Storage (GetLastError and TlsGetValue, respectively). On x64 versions of Windows, the FS register has been replaced by the GS register. Otherwise they work pretty much in the same manner.

Although this article focuses on x64 from the user mode perspective, there is one important kernel mode architectural addition to point out. New in Windows for x64 is a technology called PatchGuard, which is aimed at both security and robustness. In a nutshell, user mode programs or drivers that alter key kernel data structures such as the syscall tables and the interrupt dispatch table (IDT) create security holes and potential stability problems. For the x64 architecture, the Windows folks decided that modifying kernel memory in unsupported ways wouldn't be tolerated. The technology to enforce this is PatchGuard. It uses a kernel mode thread to monitor changes to critical kernel memory locations. If that memory is changed, the system stops via a bugcheck.

All things considered, if you're familiar with the Win32 architecture and how to write native code that runs on it, you won't find many surprises in the move to Win64. You can consider it to be mostly just a roomier environment.

Just Enough x64 to Get By

Now let's take a look at the CPU architecture itself, since a basic understanding of the CPU's instruction set makes development (especially debugging!) much easier. The first thing you'll notice in compiler-generated x64 code is how remarkably similar it is to the x86 code you know and love. This definitely wasn't the case for you folks who learned Intel IA64 coding.

The second thing you'll notice shortly thereafter is that the register names are slightly different than you're used to, and that there's a lot more of them. General-purpose x64 registers have names that begin with R, as in RAX, RBX, and so on. This is an evolution of the old E-based naming scheme for 32-bit x86 registers. Way back in the mists of time, the 16-bit AX register became the 32-bit EAX, the 16-bit BX became the 32-bit EBX, and so on. Transitioning from the 32-bit versions, all the E registers become R registers in their 64-bit incarnations. Thus, RAX is the successor to EAX, RBX succeeds EBX, RSI replaces ESI, and so on down the line.

In addition, eight new general-purpose registers (R8-R15) were added. The list of primary 64-bit general-purpose registers looks like **Figure 5**.

Figure 5

RAX
RBX
RCX
RDX
RSI
RDI
RSP
RBP
R8
R9

R10
R11
R12
R13
R14
R15

Also, the 32-bit EIP register becomes the RIP register. Of course 32-bit instructions must continue to run, so the original, smaller form factor versions of these registers (EAX, AX, AL, AH, and so on) are still available.

Lest you graphics and scientific programming gurus feel left out, the x64 CPU also has 16 128-bit SSE2 registers, which are named XMM0 through XMM15. The full set of x64 registers preserved by Windows can be found in the appropriately #ifdef'ed _CONTEXT structure defined in WINNT.H.

At any given time, an x64 CPU is operating in either legacy 32-bit mode or in 64-bit mode. In 32-bit mode, the CPU decodes and acts on instructions just like any other x86 class CPU. In 64-bit mode, the CPU has made some slight adjustment to certain instruction encodings to support the new registers and instructions.

If you're familiar with the CPU opcode encoding diagrams, you'll remember that the space for new instruction encodings was disappearing fast, and squeezing in eight new registers isn't an easy task. One way to do this was to eliminate a few rarely used instructions. So far, the only instructions I miss are 64-bit versions of PUSHAD and POPAD, which save and restore all the general purpose registers on the stack. Another way was to free up instruction encoding space was to eliminate segments entirely in 64-bit mode. Thus the life of CS, DS, ES, SS, FS, and GS come to an end. Not that many people will miss them.

With addresses being 64 bits, you might be wondering about code size. For instance, this is a common 32-bit instruction:

Copy

```
CALL DWORD PTR [XXXXXXXX]
```

Here, the X'ed portion is a 32-bit address. In 64-bit mode, does this become a 64-bit address, thereby turning a 5-byte instruction into 9 bytes? Luckily, the answer is no. The instruction remains the same size. In 64-bit mode, the 32-bit operand portion of the instruction is treated as a data offset relative to the current instruction. An example makes this clearer. In 32-bit mode, here's the instruction to call the 32-bit pointer value stored at address 00020000h:

Copy

```
00401000: CALL DWORD PTR [00020000h]
```

In 64-bit mode, the same opcodebytes call the 64-bit pointer value stored at address 00421000h (4010000h + 20000h). A little thought reveals that this relative addressing mode has important ramifications if you're generating code yourself. You can't just specify an 8-byte pointer value in an instruction. Instead, you need to specify a 32-bit relative address to a memory location where the actual 64-bit target address resides. Thus, there's an unspoken assumption that the 64-bit target pointer must lie within 2GB of the instruction that uses it. Not that big a deal for most folks, but if you do dynamic code generation or modify existing code in memory, it can byte you!

A key advantage of all the x64 registers is that compilers can finally generate code that passes most parameters in registers rather than on the stack. Pushing parameters on the stack incurs memory accesses. We've all had it drilled into our heads that a memory access that's not found in the CPU cache causes the CPU to stall for many cycles waiting for your regular RAM memory to catch up.

In designing the calling convention, the x64 architecture took advantage of the opportunity to clean up the mess of existing Win32 calling conventions such as __stdcall, __cdecl, __fastcall, _thiscall, and so on. In Win64, there's just one native calling convention, and modifiers like __cdecl are ignored by the compiler. The reduction in calling convention flavors is a wonderful boon for debuggability, among other things.

The primary thing to know about the x64 calling convention is its similarity to the x86 fastcall convention. Using the x64 convention, the first four integer arguments (from left to right) are passed in 64-bit registers designated for that purpose:

Copy

```
RCX: 1st integer argument RDX: 2nd integer argument R8: 3rd integer argument
R9: 4th integer argument
```

Integer arguments beyond the first four are passed on the stack. The this pointer is considered an integer argument, so can always be found in the RCX register. As for floating point parameters, the first four are passed in the XMM0 through XMM3 registers, with subsequent floating point parameters placed on the thread stack.

Drilling into the calling convention a bit, even though an argument can be passed in a register, the compiler still reserves space on the stack for it by decrementing the RSP register. At a minimum, each function must reserve 32 bytes (four 64-bit values) on the stack. This space allows registers passed into the function to be easily copied to a well-known stack location. The callee function isn't required to spill the input register params to the stack, but the stack space reservation ensures that it can if needed. Of course, if more than four integer parameters are passed, the appropriate additional amount of stack space must be reserved.

Let's look at an example. Consider a function passing two integer parameters to a child function. The compiler not only sticks the values in RCX and RDX, it also subtracts 32 bytes from the RSP stack pointer register. In the callee function, the parameters can be accessed in the registers (RCX and RDX). If the callee code needs the register for some other purpose, it can copy the registers into the reserved 32-byte stack region. **Figure 6** shows the registers and stack after six integer parameters have been passed.

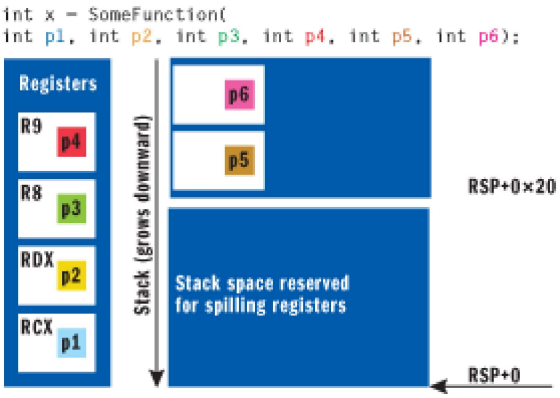

Figure 6** Passing Integers **

Parameter stack cleanup is a bit funny on x64 systems. Technically, the caller is responsible for cleaning up the stack, not the callee. However, you'll rarely see RSP adjusted anywhere other than in the prologue and epilogue code. Unlike the x86 compiler, which explicitly adds and removes parameters on to the stack with PUSH and POP instructions, the x64 code generator reserves enough stack space to call whatever the largest target function (parameter-wise) uses. It then uses the same stack region over and over to set up the parameters when calling child functions.

Put another way, the RSP rarely changes. This is quite different from x86 code, where the ESP value fluctuates as parameters are added and cleared from the stack.

An example helps here. Consider an x64 function that calls three other functions. The first function takes four params (0x20 bytes), the second takes 12 params (0x60 bytes), and the third takes eight params (0x40 bytes). In the prologue, the generated code simply reserves 0x60 bytes on the stack and copies parameter values into the appropriate spots within the 0x60 bytes so that the target functions can locate them.

A good description of the more detailed intricacies of the x64 calling convention can be found in Raymond Chen's blog . I won't belabor all the details, but here are some highlights. First, integer parameters that are less than 64-bits are sign extended, then still passed via the appropriate register, if among the first four integer parameters. Second, at no point should any parameter be in a stack location that's not a multiple of 8 bytes, thus preserving 64-bit alignment. Any argument that's not 1, 2, 4, or 8 bytes (including structs) is passed by reference. And finally, structs and unions of 8, 16, 32, or 64-bits are passed as if they were integers of the same size.

A function's return value is stored in the RAX register. The exception is for floating-point types, which are returned in XMM0. Across calls, these registers must be preserved: RBX, RBP, RDI, RSI, R12, R13, R14, and R15. These register are volatile and can be destroyed: RAX, RCX, RDX, R8, R9, R10, and R11.

Earlier I mentioned that the OS walks stack frames as part of the exception handling mechanism. If you've ever written stack-walking code, you know that the almost ad hoc nature of Win32 frame layout makes the process a tricky proposition. The situation is much better on x64 systems. If a function allocates stack space, calls other functions, preserves any registers, or uses exception handling, that function must use a well-defined set of instructions for generating standard prologues and epilogues.

Enforcing a standard way of creating a function's stack frame is one way the OS can guarantee (in theory) that the stack can always be walked. In addition to consistent, standardized prologues, the compiler and linker must also create an associated function table data entry. For the curious, all these function entries end up in table that's an array of IMAGE_FUNCTION_ENTRY64, defined in winnt.h. How do you find this table? It's pointed to by the IMAGE_DIRECTORY_ENTRY_EXCEPTION entry in the PE header's DataDirectory field.

I've covered a lot of architectural ground in a short amount of space. However, with an understanding of these big picture concepts and an existing knowledge of 32-bit assembly

language, you should be able to understand x64 instruction in the debugger within a relatively short period of time. As always, practice makes perfect.

Developing for x64 with Visual C++

Although it was possible to write x64 code with the Microsoft® C++ compiler prior to Visual Studio® 2005, it was a clunky experience in the IDE. For this article I'll therefore assume that you're working with Visual Studio 2005 and that you've selected the x64 tools, which aren't enabled in a default installation. I'll also assume that you have an existing Win32 user mode project in C++ that you'd like to build for both x86 and x64 platforms.

The first step in building for x64 is to create the 64-bit build configuration. As a good Visual Studio user, you're already aware that your projects have two configurations by default: Debug and Retail. All you need to do here is create two more configurations: Debug and Retail in their x64 guises.

Begin with your existing project/solution loaded. On the Build menu, select Configuration Manager. In the Configuration Manager dialog box, from the Active solution platform dropdown menu, select New (see **Figure 7**). You should now see another dialog entitled New Solution Platform.

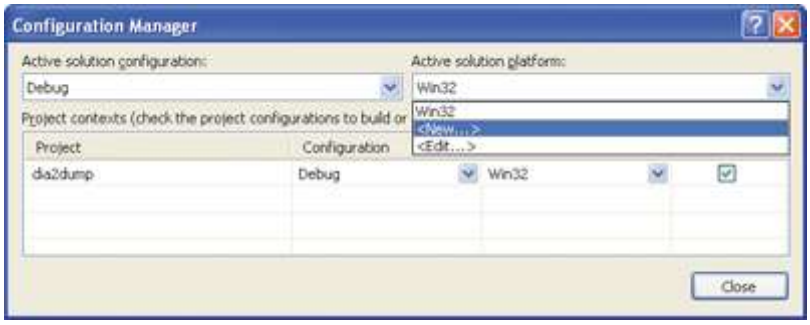

Figure 7** Creating a New Build Configuration **

Select x64 as your new platform (see **Figure 8**) and leave the other settings in their default state; then click OK. That's it! You should now have four possible build configurations: Win32 Debug, Win32 Retail, x64 Debug, and x64 Retail. Using the Configuration Manager, you can easily switch between them.

Now let's see how x64-compliant your code is. Make the x64 Debug configuration the default, and then build the project. Unless the code is trivial, odds are that you'll get some compiler errors that don't occur in the Win32 configuration. Unless you've completely forsaken all principles of writing portable C++ code, it's relatively easy to fix these issues so

that your code is truly Win32 and x64 ready, without requiring reams of conditionally compiled code.

Figure 8** Selecting the Build Platform **

Making Your Code Win64-Compliant

Probably the biggest effort in converting Win32 code to x64 is in getting your type definitions correct. Remember my earlier discussion of the Win64 type system? By using the Windows typedef types rather than the C++ compiler's native types (int, long, and so on) the Windows headers make it easy to write clean Win32 x64 code. You should continue this consistency in your own code. For instance, if Windows passes you an HWND, don't store it in a FARPROC just because it's handy and easy.

Having upgraded a lot of code, perhaps the most common and easy error I've seen is in assuming that a pointer value can be stored or transported in a 32-bit type such as an int, long, or even a DWORD. Pointers in Win32 and Win64 are different sizes by necessity, while integer types remain the same size. However, it's also not feasible for the compiler to disallow pointers from being stored in an integral type. It's a C++ habit that's just too ingrained.

To the rescue come the _PTR types defined in the Windows headers. Types such as DWORD_PTR, INT_PTR, and LONG_PTR let you declare variables that are of integral type, but that are always wide enough to store a pointer on the target platform. For instance, a variable defined as type DWORD_PTR is a 32-bit integer when compiled for Win32 and 64-bit when compiled for Win64. With practice, it became second nature for me when declaring types to ask, "Do I want a DWORD here, or do I really mean DWORD_PTR?"

As you'd expect, there might be occasions when you specify exactly how many bytes you need for an integer type. The same header file (Basetsd.h) that defines DWORD_PTR and

friends also defines size specific integers such as INT32, INT64, INT16, UINT32, and DWORD64.

Another issue related to type size differences is printf and sprintf formatting. I'm certainly guilty of using %X or %08X to format pointer values in the past, and have been bitten when I ran that code on x64 systems. The correct way is to use %p, which automatically accounts for the pointer size on the target platform. In addition, printf and sprintf have the I prefix for size-dependent types. For instance, you might use %Iu to print out a UINT_PTR variable. Likewise, if you know the variable will always be a 64-bit signed value, you could use %I64d.

Having cleaned up errors caused by type definitions that aren't Win64 ready, you may still have code that can only run in x86 mode. Or perhaps you need to write two versions of a function, one for Win32 and the other for x64. This is where a set of preprocessor macros come in handy:

```
_M_IX86 _M_AMD64 _WIN64
```

Proper use of the preprocessor macros is essential to writing correct cross-platform code. _M_IX86 and _M_AMD64 are defined only when compiling for the specified processor. _WIN64 is defined when compiling for any 64-bit version of Windows, including the Itanium edition.

When using a preprocessor macro, think hard about what you want. For instance, is the code truly specific to the x64 processor and nothing else? Then use something like:

```
#ifdef _M_AMD64
```

On the other hand, if the same code could work on both x64 and Itanium, you might be better off with something like:

```
#ifdef _WIN64
```

A convention that I've found useful is that whenever I use one of these macros, I always explicitly create the #else cases so that I know early if I've forgotten something. Consider the following badly written code:

```
#ifdef _M_AMD64 // My x64 code here #else // My x86 code here #endif
```

What happens if I now compile this for a third CPU architecture? My x86 code will unintentionally be compiled. A much better way to phrase the previous code is like this:

```
#ifdef _M_AMD64 // My x64 code here #elif defined (_M_IX86) // My x86 code here
#else #error !!! Need to write code for this architecture #endif
```

The one part of my Win32 code that didn't port easily to x64 was my inline assembler, which Visual C++ doesn't support for the x64 target. Fear not, assembler heads. A 64-bit MASM (ML64.exe) is provided and is documented via MSDN. ML64.exe and other x64 tools (including CL.EXE and LINK.EXE) are available from the command line. You can just run the VCVARS64.BAT file, which adds them to your path.

Debugging

You've finally gotten your code to compile cleanly on Win32 and x64 builds. The final piece of the puzzle is running and debugging it. Regardless of whether you build your x64 version on an x64 box, you'll need to use the Visual Studio remote debugging features to debug in x64 mode. Luckily, if you're running the Visual Studio IDE on the 64-bit machine, the IDE takes care of all of the following steps for you. If, for some reason, you can't use remote debugging, your other option is to use the x64 version of WinDbg . However, you'll be giving up many of the debugging niceties found in the Visual Studio debugger.

If you've never used remote debugging, there's not much cause for concern. Once you get it set up, remote debugging can be as seamless as local.

The first step is to install the 64-bit MSVSMON on the target machine. This is typically done by running the RdbgSetup program that comes with Visual Studio. Once MSVSMON is running, use the Tools menu to configure the appropriate security settings (or lack thereof) for the connection between your 32-bit Visual Studio and the MSVSMON instance.

Next, within Visual Studio you'll want to configure your project to use remote debugging for x64 code, rather than attempting local debugging. You can start this process by bringing up the project's properties (see **Figure 9**).

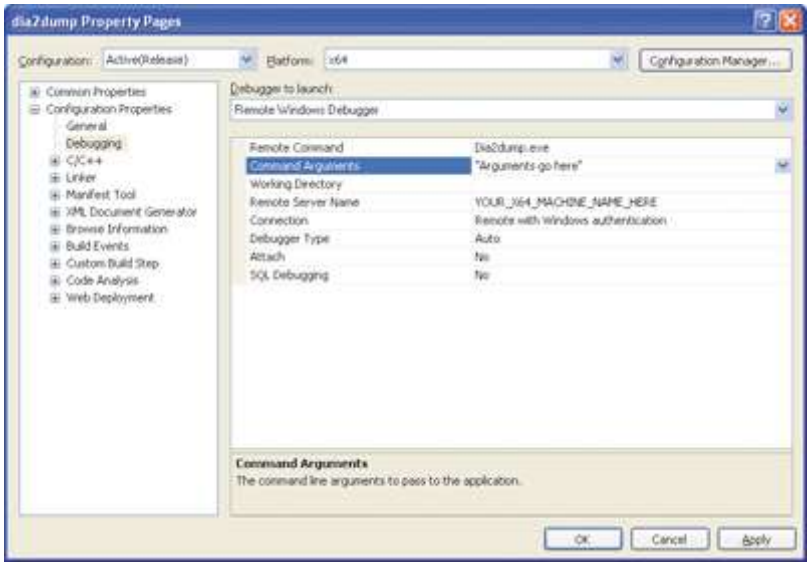

Figure 9** Debugging Properties **

Make sure your 64-bit configuration is current and select Debugging under Configuration Properties. Near the top is a drop-down menu titled Debugger to launch. Normally this is set to Local Windows Debugger. Change this to Remote Windows Debugger. Below that, you can specify the remote command to execute when you start debugging (the program name, for example) as well as the remote machine name and connection type.

If you set up everything properly, it's possible to start debugging your x64 target application in the same way you start Win32 apps. You'll know you've successfully connected to the MSVSMON because its trace window emits a "connected" string each time the debugger successfully attaches. From here, it's mostly the same Visual Studio debugger that you know and love. Be sure to bring up the registers window and look at all those glorious 64-bit registers, and pop into the disassembly window to check out that oh-so-familiar-but-just-a-little-different x64 assembly code.

Note that a 64-bit minidump can't be loaded directly into Visual Studio like 32-bit dumps. Instead, you'll need to use the Remote Debugging. Also, interop debugging between native and managed 64-bit code isn't currently supported in Visual Studio 2005.

What About Managed Code?

One of the great things about coding with the Microsoft .NET Framework is that much of the underlying operating system is abstracted away for general-purpose code. In addition,

the IL instruction format is CPU agnostic. As a result, at a theoretical level, it should possible for a .NET-based program binary built on a Win32 system to run unmodified on an x64 system. The reality is a little bit more complicated.

The .NET Framework 2.0 comes with an x64 version. After installing this on my x64 machine, I was able to run the same .NET executables that I'd previously run on my Win32 box. How cool is that? Of course there's no guarantee that every single .NET-based program will run equally well on Win32 and x64 without a recompile, but it does "just work" a reasonable percentage of the time.

If your managed code explicitly invokes native code (for instance, through P/Invoke in C# or Visual Basic®), you will very likely run into trouble if you try to run against the 64-bit CLR. However, there is a compiler switch (/platform) that allows you to be more specific about which platform your code should run on. For instance, you might want your managed code to run in WOW64, even though a 64-bit CLR is available.

Wrap Up

All things considered, moving to an x64 version of Windows was a relatively painless experience for me. Once you have a good grasp of the relatively minor differences in the OS architecture and tools, it's easy to keep one code base running on both platforms. Visual Studio 2005 makes the effort substantially easier. And more x64-specific versions of device drivers and tools such as Process Explorer from SysInternals.com are appearing every day, so there's no reason not to jump in!

Matt Pietrek has co-written several books on Windows system-level programming as well as the Under the Hood column for *MSDN Magazine*. Previously he was a primary architect for the NuMega/Compuware BoundsChecker series of products. He now works on the Visual Studio team at Microsoft.