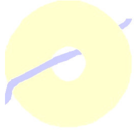

TMurgent Technologies

Porting C++ Applications to x64

TMurgent Developer Series

White Paper by Tim Mangan
Founder, TMurgent Technologies
April, 2006

 Copyright 2006 TMurgent Technologies

Introduction

This White Paper provides a quick primer on what a developer needs to know when faced with the task of porting a 32-bit program to run in 64-bit mode on an "x64" processor. The information in this paper is based on experience in porting both GUI based applications, as well as *Windows* based Services.

The "x64" platform includes both *AMD* and *Intel* chipsets. AMD introduced the x64 technology in their *Opteron* line and Intel offers it in some of their *Xeon* lines. These CPUs can run 64 or 32 bit Operating Systems.

Intel also has an older 64-bit CPU called *Itanium* that only runs in 64-bit mode. This paper does not cover Itanium, although many of the points in this paper will also be relevant in porting to Itanium. Intel has signaled that Itanium will continue to be supported, although market adoption has been quite limited.

Reasons for Porting

To start with, you might not need to port. Many companies have been purchasing x64 CPUs but running with the 32-bit version of the *Microsoft* Operating System. Microsoft released an x64 version of their operating systems in 2005 after a long beta cycle. However, due to a lack of device drivers many companies could not run x64 version, and continue to use the 32-bit OS[1]. To those customers an x64-bit port would be useless.

However, x64 drivers and Anti-virus products are now appearing in the market and selecting the x64 OS is becoming more reasonable. The primary drivers for shifting to x64 OS are very large programs (such as databases) and Terminal Servers. The companion White Paper, "x*64 Servers: Do you want 64 or 32 bit apps with that server"*[2] addresses why an enterprise might choose to install 32-bit applications over x64 applications on their x64 Terminal Server, even if x64 is available.

---

[1] 32-bit mode is also referred to as "x86", which refers to the generic version of Intel's PC CPU architectures starting with the 8086 all the way through the current 32-bit lines. In reality, many references to x86 mean a subset of those CPUs starting with the Pentium.
[2] Available on-line at http://www.tmurgent.com/WhitePapers/WP_x64ShouldYou.pdf.

 Copyright 2006 TMurgent Technologies

A port to x64 is necessary in the following cases:

- The software contains a kernel mode driver. All kernel components running in the x64 OS must be 64-bit. No exceptions, unless you can re-write the 32-bit driver as a User-mode driver.
- The software uses some of the under-documented "Native API" calls into the kernel that are not supported by 32-bit applications in an x64 OS.
- The software needs access to more than 2GB or user mode memory[3].
- The software needs to access a component (dll or such) that only exists as a x64 executable[4].
- Somebody thought it was a good idea. The boss signs your paycheck; the customer puts money into the company bank account to cover the check.

A 32-bit program that gains no added functionality by being a 64-bit program will be larger. There are some CPU efficiencies available if you port. For example, although 32-bit instructions are executed as efficiently via hardware (not software emulation), there are some API overhead incurred when the operating system redirects activities such as registry access. These are very small overheads and not typically a reason to port. The added registers when running in 64-bit can make CPU intensive applications more efficient also (discussed later in this paper).

Also note that the x64 OS does not support 16-bit applications. So at a minimum those need to be rewritten to 32 bit (not covered in this paper). But don't forget about the installers! Many companies continue to use old installers that are 16-bit. These will not work either and you need to update your installer.

Some Facts

In a 32-bit processor, unless you use PAE, you are limited to 4GB of memory, which is 2^{32}. 2^{64} would be 16 Exabytes, where an Exabyte is a billion Gigabytes. However the current x64 CPUs run with only 40-bits of the 64-bits for addressing used in a 64-bit container. 2^{40} is 16 Terabytes, which should be enough memory to last us a while. It is possible that future CPUs may increase the number of address bits, but certainly not for quite a while. I have seen documentation that claims the current architecture can extend to use 52 bits (providing a

[3] Other solutions such as PAE might also solve this problem on a 32-bit OS.
[4] We are unaware of any at this time, but you know it is inevitable.

range of 4 Petabytes), implying (but not stating unequivocally) that the OS is 52-bit ready if the CPUs show up. Your application code need not (indeed **should not**) worry about the number of bits used in memory addressing – just treat them as 64-bit quantities and everything will be good. By the way, a page of memory in x64 is 4 Kilobytes, just as it was in 32-bit mode.

Unlike the shift from 16-bit to 32-bit, not everything doubles in size. Obviously pointers, which were 32-bits are now 64-bit. Integers, however, usually remain unchanged. We have had both 32 and 64 bit integers available for some time. For most applications, 32-bits is still big enough for whatever you are counting. Floating point numbers also remain the same size. Handles (including HWNDS, IO and timer handles) remain 32-bits as they are really implemented as indexes in lookup tables that don't need to grow. Size information normally should double from 32 to 64 bits also, and as is discussed later in this paper is probably the trickiest of the problems in porting.

The operating system provides some redirections for applications running in 32-bit mode to ease your pain. This is called the WOW64 subsystem. There are many ways that these redirections appear, and the methods used are inconsistent in naming convention which leads to confusion.

For example, 32-bit programs need to load 32-bit dlls; 64-bit programs need to load 64-bit dlls. This includes all those system dlls. On an x64 OS, the *system32* contains the x64 versions of these dlls. Microsoft chose to not rename this folder to system64 due to programs, but to redirect. So a 64-bit program requesting a system dll will search the actual system32 folder and receive a 64-bit dll. A 32-bit program, however, will be redirected by the WOW64 subsystem to a different folder called SysWow64 (as in C:\Windows\SysWow64). This redirection takes place whether you call LoadLibrary or open a file reference, including a hard-coded "C:\Windows\system32" (which you shouldn't do, but it manages to work. You should use environment variables for things like the system root, windows root, system32, program files, and user profile).

Similarly, a redirection occurs for the Program Files folder. 64-bit programs go into the *Program Files* folder and 32-bit programs go into *Program Files (x86).* Again, redirection is automatic. 32-bit programs, both exes and runtime components with extensions such as dll and ocx, use a file format called PE32. 64-bit programs use an extension to this format called PE32+. The format is nearly identical

except for header settings and a few places where pointers and offsets are used.

WOW64 also does a redirection. HKLM/Software contains keys for use by 64 bit applications. 32-bit applications get pointed to a sub-key HKLM\Software\Wow6432Node. This includes entries for 32 bit applications as well as the classes sub-key which is used by COM based applications and ensures that 32-bit programs get the right COM object. Note that HKCU does not have a redirected sub-key for WOW64, nor (to my knowledge) does HKLM/SYSTEM.

Another confusing aspect about all this redirection is that you need to be careful about how you look for files as a developer. For example, there is both a 64 and 32 bit command shell (cmd.exe) available. Normally you use the 64-bit version that will show you files in their normal places (system32 contains 64 bit dlls and SysWow64 contains the 32 bit ones), but if you run the 32-bit shell the redirection occurs. Similarly there is both a 32 and 64 bit version of the Internet Explorer. When using 3rd party tools this may lead to confusion if you don't pay attention.

Microsoft does offer some escape clauses to all this redirection as well. IsWow64Process is a new runtime library function that will tell your program if it is 32/64 bits. Other methods can "work around" the WOW64 when needed as well. Rather than get bogged down in that detail, look into the MSDN documentation if you really need to cross access.

What It Takes

First of all, you don't really need an x64 system to build 64 bit code. Sure, you need one for testing, but it doesn't need to be your development machine. While this has nothing to do with x64, we use Virtual Machines for all our development projects these days.
If you haven't invested in VMs you should consider it. It makes a very nice, portable, and reproducible environment for building software. But I digress. By the way, you can also debug 64 bit code from a 32-bit machine if you use remote debugging. You need to run the RdbgSetup from your VS2005 disk on the target x64 machine, then you can use the VS2005 IDE to connect up (be sure to select the x64 platform in the IDE).

While it is possible to build x64-bit applications using other tools, moving to Visual Studio 2005 is well advised. Intel also makes nice

 Copyright 2006 TMurgent Technologies

compilers and if you are not doing Win32 GUI based apps (the API is still called Win32 even if you are writing a 64-bit program) that might be an option. The remainder of this white paper assumes that you are using VS2005.

The first step is to port the application to VS2005 and get the application compiling, linking, and running with that tool set in 32-bit mode. Another white paper in the *TMurgent Developer Series*, "*Moving to Visual Studio 2005*"[5], covers porting applications from Visual Studio 2003 to 2005.

Once you have the 32-bit building and running, it is time to take on a 64-bit build. You probably already have two build configuration options for your applications – namely Debug and Release. When you look at your project properties you will see the familiar Build selection, plus a new selection called "*Platform*". Your converted project will already have one defined platform called "Win32", and should look similar to Figure 1.

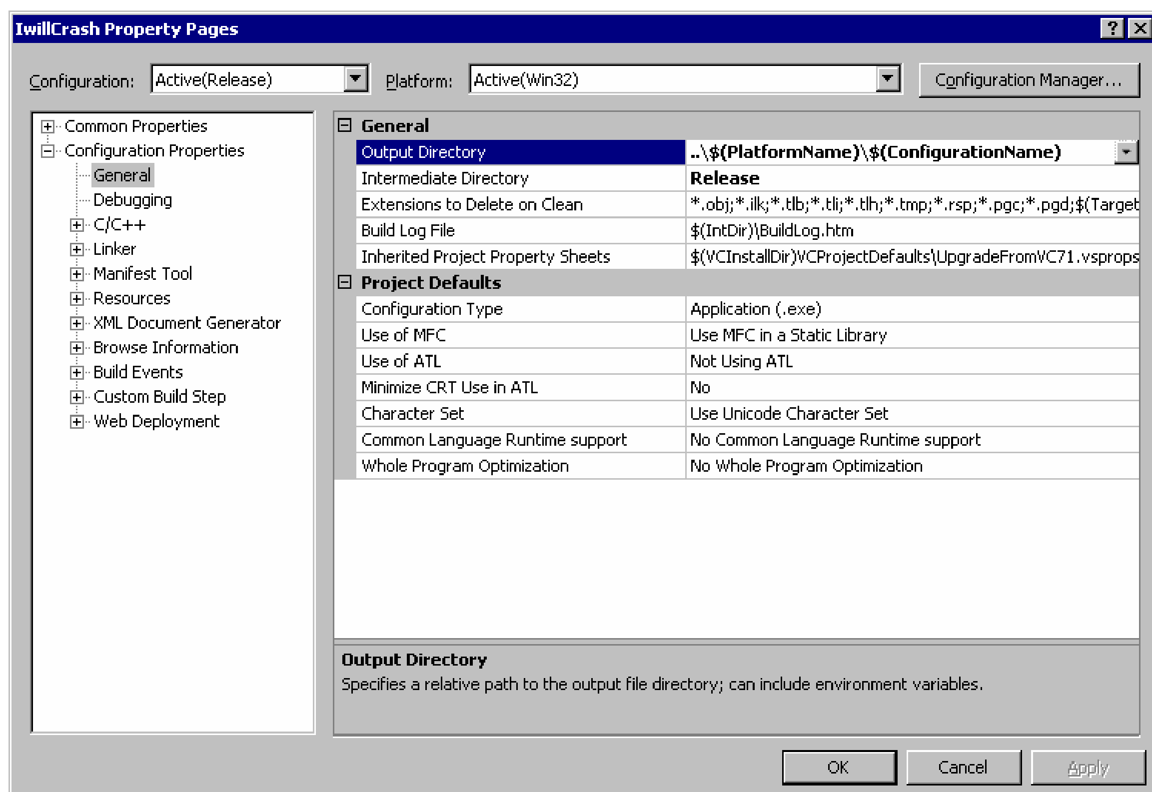

Figure 1 - Project Property Page

[5] Available online at http://ww.tmurgent.com/WhitePapers/WP_VS2005.pdf.

Next you will create a new platform for Win64. You first create the platform for the solution, and then for (each) project. Click on the "Configuration Manager…" button. You will see a new dialog box like Figure 2.

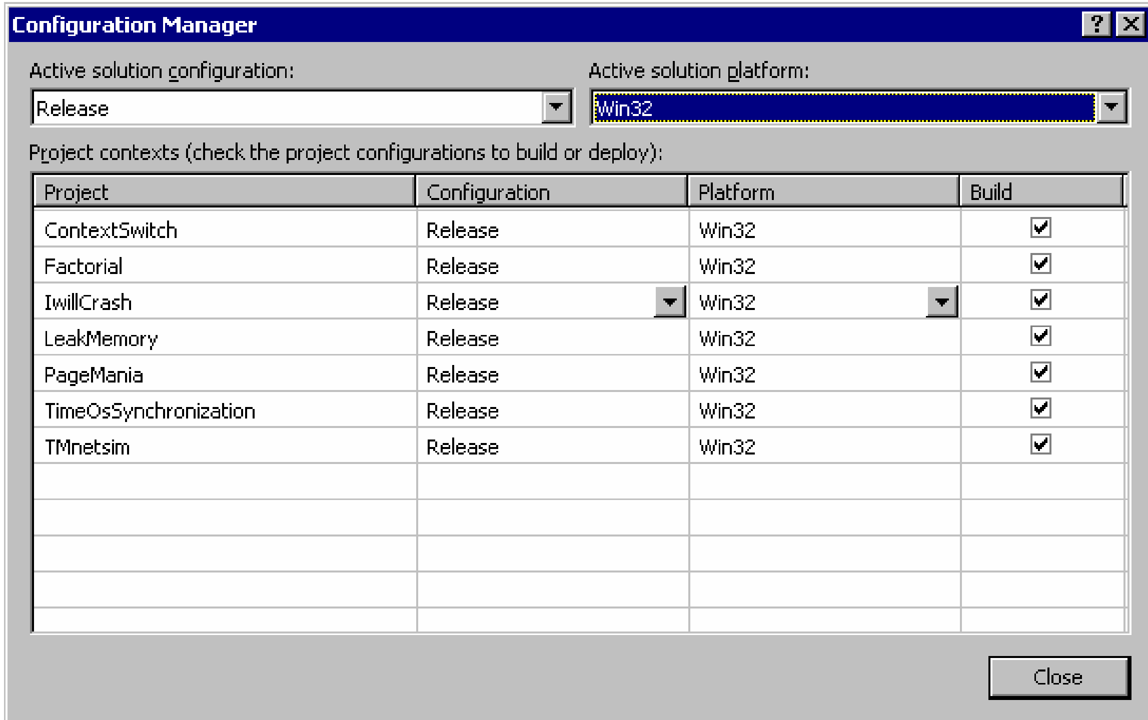

Figure 2 - Configuration Manager

Use the pull-down menu under "Active Solution Platform" and select "New". A third pop-up dialog appears similar to Figure 3.

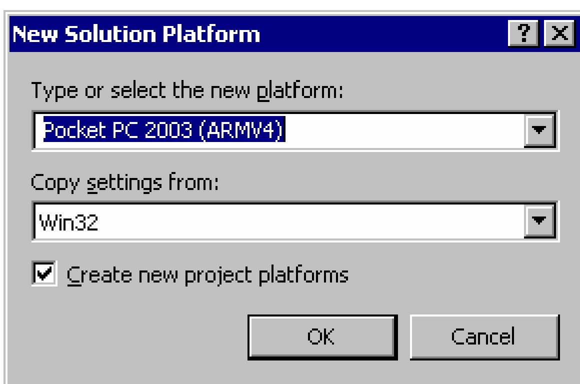

Figure 3 - New Solution Platform

You will want to select a Win64 platform (don't ask me why the ARM Pocket PC is the default here!), created as a copy of the Win32 platform with the checkbox selected to create the project platforms for

you. After you click OK you can close the Configuration Manager window also.

I typically change the general output directory for all configurations and all platforms to "..\$(Platform)\$(Configuration)". At first, I was adding 32 or x64 to the name of output executables to help me keep them straight when they get copied around; however with experience I discarded that practice.

Most of your code will probably be OK under x64 bit, especially if you followed reasonable coding standards. The biggest problem I find is the incorrect use of placing size information into a DWORD and passing that as a parameter. Since size_t grows from 32 to 64 bits, if your code uses a DWORD you may need to make adjustments.

The next most frequent problem is structure alignment issues. Programs that communicate with other programs via a socket or binary file may need to consider the case of dealing with a counterpart that is either 32 or 64 bit.

It is very rare that I need code that is specific to a platform. Sometimes the new IsWow64Process () runtime method will do the trick, but sometimes you need to #ifdef.

One example is in dealing with XML parsing. If you do XML parsing on x64 you really want to use the msxml6 libraries. As of this writing, there is no SDK for that version so you need to directly reference the appropriate dll file for the build to get the COM interfaces that you need.

```
#ifdef _WIN64
   #import "..\..\msxml\msxml6_x64\msxml6.dll" raw_interfaces_only
#else
   #import "..\..\msxml\msxml6_32\msxml6.dll" raw_interfaces_only
#endif
```

Figure 4 - Example of platform ifdef

I should mention that for those particular lines to work you need both the 32 and 64 bit versions of the dll files on you system. That means you must install MSXML6 on both a 32-bit and 64-bit OS and copy them to your development system.

Finally, once everything is up-and-running you can consider the performance aspects of x64. I ran several builds of a small test program on an x64 server to show the differences in memory size and handle usage:

Example	Exe Size	Working Set	Page	Handles
VS2003 built 32-bit	264KB	3660KB	1244KB	27
VS2005 built 32-bit	320KB	3612KB	1228KB	27
VS2005 built 64-bit	478KB	4976KB	2628KB	20

Figure 5 - Size and Handles of different builds

X64 programs are larger. Some of this is the increased pointer sizes, some would be larger instructions. While an x64 server allows you to install and use more memory, you still have to buy that memory. Notice the difference in handles. Part of the Wow64 overhead is the use of those handles. Handles are not normally a performance issue, but it is interesting to notice.

On the flip side of the performance equation, having many more registers to deal with, x64 programs can behave with greater efficiency. In particular you may want to look at function call parameters, as up to 4 64-bit integers or addresses may now be passed in registers rather than the stack[6]. Additional integer parameters must be placed on the stack. This saves the additional CPU instructions to push and pop values onto the stack (although stack space is allocated for the parameters in case the called function needs the register). For frequently called functions in CPU intensive applications this may lead to performance optimizations by modifying calling parameters.

Summary

This paper provides a quick introduction to porting C++ programs to the x64 platform. As a rule of thumb, I find that porting from VS2003 to VS2005 is more work than porting to x64 for most software applications.

For detailed information on points covered (and not covered) in this paper I suggest that you consult MSDN-online documentation.

[6] If there is a hidden "this" parameter this is placed in the RCX register leaving three registers for integer parameters.

 Copyright 2006 TMurgent Technologies

About TMurgent Developer Series White Papers

At TMurgent, we help Software vendors with development issues, especially when it comes to system performance, management, and working with Terminal Services. Recent projects include helping companies make their products more Terminal Services aware, kernel interfacing, and porting to VS2005 and x64 platforms. We provide education to the developer community via the "*TMurgent Developer Series*" white papers to advertise our services. Please visit our website at www.tmurgent.com for more information.

 Copyright 2006 TMurgent Technologies