

Preparing Code for the IA-64 Architecture (Code Clean)

A Programmer's Reference

intel[®]

Table of Contents

About This Guide	3
Section 1: Introduction	4
Preparing Code for the Intel® IA-64 Architecture	4
Porting Considerations	5
Compiler Flags	5
Eliminating In-line Assembly	6
Section 2: Data Models for the Intel® IA-64 Architecture ...	7
UNIX* and Windows* Data Models	7
New Explicitly Sized Data types	7
New Scalable Data Types	9
Using Code Guards	10
Big Integer Declarations	11
Windows* 2000 (64-bit) Window-Class Data	12
Using Formatted I/O Messages	13
Section 3: Porting Pointers	14
Type-casting Pointers	14
Pointers to Legacy Data	15
Accessing Data Structures With an Offset	15
Section 4: Preparing for Multi-platform Compilation	17
Using a Compatibility File	17
File-Naming Conventions	17
Directory Structure	18
Build Rules	18
Section 5: Calling Conventions	20
Function Call	20
Function Call Through General Registers	20
Function Return Through Floating-Point Registers	21
Explicit Prototyping	21
Explicit Prototyping of Floating-Point Functions	22
Section 6: Manual Code Cleaning	24
Managing Data Size	24
General Rule	24
Structure Padding	24
Pointer Cleaning	25
Hash Algorithms	26
Section 7: Miscellaneous Topics	27
Jump Buffer	27
Thread Stacks	27
UNIX Thread Stack	27
Windows Thread Stack	28
Context Structure	29
UNIX Context	29
Windows Context	29
Constants	30
Integer-Constant-Type Suffixes	30
Hex Constants	30
Truncation of long via doubles	31
Section 8: Checklist	32
Section 9: Additional Resources	33
Section 10: Intel® IA-64 Architecture Features	34
Glossary	35

About This Guide

This guide is intended for programmers who are ready to prepare programs previously written for the Intel® IA-32 architecture in Microsoft Visual C/C++* or ISO/ANSI C/C++ for the new Intel® IA-64 architecture.

This guide will help you migrate an existing UNIX* or Windows* application running on the Intel IA-32 architecture to an application for the Intel IA-64 architecture.

The following topics will be covered in this guide:

- The changes in the Intel IA-64 data model for both UNIX and Windows* 2000 (64-bit).
- How these changes might affect your code and some recommendations to successfully port your code.
- Recommendations on porting code to *both* environments.

This guide will not cover the following:

- General Windows or UNIX programming information.
- 16-bit or DOS.
- Intel IA-64 instruction set or architecture or other IA-64 details not related to porting. But we'll show you where you can find more information.

This guide covers the steps involved to enable your software to take advantage of the 64-bit addressability of IA-64. Some applications may not require all the preparation described in this guide, while others may have issues not covered here.

Use this guide as you prepare for porting and during the porting process to help you identify areas of your code that might need to be changed and areas that might need to be looked at more closely.

To help effectively port to the Intel IA-64 architecture, your existing application should already successfully compile and run under Intel IA-32 architecture. In addition, you'll need the following:

- The complete source code you'll be porting.
- An Intel IA-64 architecture development environment.
- Adequate test cases to thoroughly exercise your code to ensure high-code coverage.

For more information about the Intel IA-64 architecture and related tutorials and subjects, see **Section 9, Additional Resources** at the end of this guide.

To help you with the preparation process, refer to **Section 8, Checklist** for a checklist of tasks related to preparing your code for the Intel IA-64 architecture.

For terms that may be unfamiliar, see the **Glossary** at the end of this guide.

Section 1: Introduction

This section includes the following general information regarding porting your code:

- The porting process
- Important porting considerations
- Compiler flags
- In-line assembly

Preparing Code for the Intel® IA-64 Architecture

With the development of the Intel® IA-64 architecture, the UNIX* and Windows* 2000 (64-bit) data models have changed with new data types and 64-bit pointers to accommodate the new architecture. However, the UNIX and Windows data models have diverged. With the Intel® IA-32 architecture, both UNIX and Windows shared a common data model, and basic data types were identical. This is no longer the case, making porting a little more challenging. These changes and how they can impact your code will be described in this guide.

To prepare your code for the IA-64 architecture, you'll need to consider the following:

1. Understand the changes in the data models.
2. Select an appropriate porting model, which includes naming conventions, directory structure, proper use of data types, and conditional compilation macros.
3. Characterize your code to help you identify areas that might need to be changed. Here are some things to look for in your code.
 - Whether or not the code uses ANSI C/C++ features. In most instances, ANSI C/C++ makes porting to both UNIX and Windows easier; however, there is no ANSI C/C++ standardization for addressing 64-bit architectures yet. Where possible, it is better to use ANSI C/C++.
 - The kinds of data types being used. Both UNIX/64 and Windows 2000 (64-bit) introduce new data types. You'll need to evaluate your code for the appropriate and most efficient use of these types.
 - External data structures accessed by the application. With the Intel IA-64 architecture, natural alignment of data is critical. You may need to re-arrange fields or make other changes that make your code more efficient in the 64-bit world and reduce data bloat.
 - Hard-coded "size" values. These are problematic, because offsets and sizes may have changed with the new architecture. What used to be 4 bytes may now be 8 bytes.
 - Header file usage. Header files are a must. They contain definitions and declarations which assure identical interpretations of type names, function call rules, machine-specific identifiers, and data structures by function library providers and their users.
 - Data structures that are used often and routines that are called frequently. These are programming features that will benefit most from careful scrutiny to assure minimal memory usage overhead, and efficient parameter passing and register usage.
4. Do a test compile for your source with a tool capable of checking for 64-bit type mismatch problems. Windows compilers implement this with the "/Wp64" option, and a 64-bit "lint" is available for most UNIX OSes.
4. Revise your code. This is the real work.
5. Compile the code with 32-bit and 64-bit compilers to assure that one body of source code can successfully build both iA64 and iA32 executables.
6. Run regression tests on both executables. This subject is beyond the scope of this document.

During your porting effort, static and dynamic analysis can help reduce the work involved and pinpoint areas where you can get more benefit for your effort. Static analysis can help immediately find data type mismatches, and can suggest how often structures are referenced and how many times functions are called. You can use compiler functions, such as header-nesting lists and warning lists, and also employ a source code browser to trace *definers* and *users* of variables and structures.

Dynamic analysis tools can help you identify frequently-used functions and data structures that, when optimized, best benefit your application's capacity and performance. There are several runtime analysis application tools available.

Porting Considerations

There are a few general considerations to be aware of when porting to the Intel IA-64 architecture.

- For known OSes, binaries for both Intel IA-64 and Intel IA-32 architectures cannot exist within the same process space. To access 32-bit modules from an Intel IA-64 binary, you will need to use an out-of-process Inter-process Communication (IPC) technique. Some examples are shared memory, sockets, named pipes, asynchronous file access, and RPC (or derivatives like COM or DCE). IPC techniques are beyond the scope of this guide.
- All device driver code for Windows 2000 (64-bit) must be ported to the Intel IA-64 architecture. Windows 2000 (64-bit) does not support IA-32 device drivers.

Compiler Flags

This guide focuses on updating your code to achieve full 64-bit compatibility. 64-bit compilation is the default for compilers for Windows 2000 (64-bit). However, some compilers and OS runtimes additionally support 32-bit pointers and/or 32-bit virtual address space using the Intel IA-64 architecture instruction set. Table 1 lists the compilation options for IA-64 compilers in Windows.

Table 1

Microsoft* Compiler-supported Flags for 64-bit Compilation

	Default Pointer Size	Address Space	Warnings
64-bit flags (default)	Directs the assembler to generate code using a 64-bit addressing model.	/As64	/W3 most warnings /W4 verbose warnings /Wp64 perform 64-bit truncation and expansion checks.
32-bit flags (special)	Pointer truncation or extension is handled in the link step.	/As32	/W3

For UNIX compiler flags, refer to your UNIX documentation.

Eliminating In-line Assembly

Neither the Intel nor Microsoft* compilers for IA-64 support in-line assembly. Where possible, rewrite your assembly code in a higher level language, such as C or C++.

If you have to use assembly code, rewrite the code in the Intel IA-64 architecture assembly and store it in a separate file. Then perform a call to the routine. This has the advantage of carefully isolating non-portabilities from common, single source code and cleanly preparing for other platforms requiring assembly (like IA-32).

The Intel IA-64 architecture uses a new assembly language; IA-32 assembly files are not compatible with IA-64 assemblers.

Section 2: Data Models for the Intel® IA-64 Architecture

This section describes the changes in the UNIX*/64 and Windows* 2000 (64-bit) data models.

UNIX* and Windows* Data Models

The UNIX/64 and Windows 2000 (64-bit) data models have slightly diverged, differing on some data types. They still share the same meaning for the types `char`, `short`, `int`, `float`, and `double`. However, in Windows 2000 (64-bit), `long` is still 32 bits, while in UNIX/64 `long` is 64 bits. In addition, there is no ANSI C/C++ standardization for a 64-bit model, as of yet.

Here are some additional facts about the models.

In both the UNIX/64 and Windows 2000 (64-bit) data models,

- Abstract types are defined in terms of basic types, which means that when you use abstract types you ensure that parameters and structure fields always contain the correctly sized data for 32- or 64-bit compilation.

In the UNIX/64 data model,

- `int` is 32 bits; `long` and pointers are 64 bits.
- There are some new explicitly sized types.
- There are a few new functions.
- The scalable, biggest architecture type is `long`.

In the Windows 2000 (64-bit) model,

- It's called a Uniform Data Model (UDM). When your code is updated according to this model, it will run on all Windows platforms.
- `int` and `long` are 32 bits; `__int64` (new type) and pointers are 64 bits.
- Abstract types are identical for 32 and 64 bit, simplifying cross-compilation for both Windows environments.
- There are explicitly sized and scalable types.
- Some types are “upgraded” and can be used in Win32 sources also.
- The scalable, biggest architecture integral type depends upon the platform. Microsoft recommends using conditional compilation of either `long` or `__int64`; a pre-processor macro can be defined to simplify this complication..
- Some functions have been revised due to polymorphism and some other Win32 legacy functions are being obsoleted.

This section describes the new data types and how they might affect your code.

New Explicitly Sized Data types

Both the UNIX/64 and the Windows UDM provide new fixed-length, or explicitly sized, data types. These data types are defined in header files that must be included for either UNIX/64 or Windows 2000 (64-bit).

Using explicitly sized types can be helpful to clarify code purposes and make maintenance easier. However, since these types do not automatically scale with the target architecture, they should not be used for pointers, since pointers in the Intel® IA-64 architecture are commonly 64 bits.

UNIX/64 and the Windows UDM have introduced different names for fixed-length types. Fortunately, ANSI standard names, such as `short` and `int`, can still be used for some of these types, which makes it easier to create cross-compilable code for both UNIX and Windows and avoids conditional compiling. When possible, it is best to use ANSI types. However, ANSI has no standardized, explicitly sized 64-bit types.

Table 2

New UNIX*/64 Explicitly Sized Data Types¹

Architecture	New Data Types	Old Data Types
64-bit ²	<code>int64_t</code> , <code>uint64_t</code>	<code>long long</code>
32-bit	<code>int32_t</code> , <code>uint32_t</code>	<code>int</code> , <code>unsigned int</code>
16-bit	<code>int16_t</code> , <code>uint16_t</code>	<code>short</code> , <code>unsigned short</code>
8-bit	<code>int8_t</code> , <code>uint8_t</code>	<code>char</code>

¹ To use these data type in your interfaces, include `<inttypes.h>` in your source file.

² On platforms that do not yet support 64-bit integers, the compiler will generate a “missing” or “not defined” error.

Table 3

New Windows* 2000 (64-bit) Explicitly Sized Data Types³

Architecture	New Data Types	Old Data Types
64-bit	<code>INT64</code> , <code>UINT64</code> , <code>LONG64</code> , <code>ULONG64</code> , <code>DWORD64</code>	<code>__int64</code> , <code>unsigned __int64</code>
32-bit	<code>INT32</code> , <code>UNINT32</code> , <code>LONG32</code> , <code>ULONG32</code> , <code>DWORD32</code>	<code>int</code> , <code>unsigned int</code>
16-bit		<code>short</code> , <code>unsigned short</code>
8-bit		<code>char</code>

³ To use these data type in your interfaces, include `<basetsd.h>` in your source file.

⁴ `INT64` is available, but only if you are using a recent Microsoft* Platform SDK with your compilation environment.

Notice that with Windows you won't be using just `INT`, `LONG`, and `DWORD` any more. You'll need to specify the length if you want to use an explicitly sized type.

Remember

- Whenever you can, use types that mean the same thing in both UNIX and Windows.
- Explicitly sized types can help clarify code purposes for maintenance.
- The largest, efficiently handled integral type in UNIX/64 is `long`; in Windows 2000 (64-bit), it is `__int64`.
- The types `short` and `int` retain their 16- and 32-bit meanings for both UNIX and Windows OSes.
- `signed` and `unsigned` variants are also available in both environments.
- `char` is always an 8-bit type.
- Make sure you include the appropriate header file.

New Scalable Data Types

Scalable, or pointer precision, data types are polymorphic and adjust to the size of the architecture – and thus the pointer size – during compilation. If you use one of these types, it will compile with a 32-bit size under Intel® IA-32 architecture and with a 64-bit size under Intel IA-64 architecture. You won't need to conditionally compile for these types. For additional, specific information about porting pointers, see **Section 3, Porting Pointers**.

UNIX/64 and Windows 2000 (64-bit) have their own respective scalable types. However, there are also some ANSI types available in both environments as shown in Tables 4 and 5. The examples that accompany the tables might help you identify how to use the new types.

Table 4

New UNIX*/64 Scalable Data Types

Integral data types that may contain a type-cast pointer	intptr_t, uintptr_t
Integral data types intended to always contain counting numbers	long, size_t, ssize_t

Carefully check your code when you use these types. Don't mix them with fixed types. In a 64-bit environment, any value passed to an `int`, `unsigned int`, or fixed type less than 64-bits will be truncated.

Example⁵

Current Code	IA-64 Code
<pre>#include <string.h> int len; char *s; ... len = strlen(s);</pre>	<pre>#include <string.h> size_t len; char *s; ... len = strlen(s);</pre>

⁵ Red bold text in the examples indicates changed code.

Discussion

The size of `len` was changed from `int` to `size_t` to be large enough to carry pointer-sized values regardless of the architecture the code will be compiled for. This ensures that the value that `strlen()` returns will not be truncated.

Table 5

New Windows* 2000 (64-bit) Scalable Data Types

Integral data types that may contain a type-cast pointer	ANSI	intptr_t, uintptr_t
	Windows 2000 (64-bit)	LPARAM, WPARAM, LRESULT, INT_PTR, UINT_PTR, DWORD_PTR, LONG_PTR, ULONG_PTR
Integral data types intended to always contain counting numbers	ANSI	size_t, ssize_t
	Windows 2000 (64-bit)	int3264, SIZET, SSIZE_T

Carefully check your code when you use these types. Don't mix them with fixed types. In a Windows 2000 (64-bit) environment, any value passed to a DWORD, ULONG, INT, LONG or fixed type less than 64-bits will be truncated.

Many Win32 APIs now use these new types, and your code will need to adapt to them.

Notice that there are ANSI versions of these scalable types. Use them when ever possible.

Example

Current Code

```
LRESULT WINAPI MainWndProc (HWND hWnd , UINT message, UINT wParam, LONG lParam)
```

IA-64 Code

```
LRESULT WINAPI MainWndProc (HWND hWnd , UINT message, WPARAM wParam, LPARAM lParam)
```

Discussion

To ensure that the values returned by wParam and lParam are not truncated when the application is compiled for the IA-64 architecture, the types were changed to scalable types – WPARAM and LPARAM.

Remember

- Use a scalable data type when architecture-based values, such as from pointers, must be passed.
- Don't mix scalable and explicitly sized types.
- Use ANSI versions of scalable types when possible.
- Some Win32 APIs use these new types now, and your code will need to be updated with them.

Using Code Guards

Your source code may need local customization based on the target platform or architecture. In C and C++, conditional compilation is used to guard, or choose, platform-specific blocks of code, so the code is compiled only for a particular target. Code guards are expressions that use pre-processor macros and values. Both the UNIX and Windows development environments establish predefined macros you can use in conditional definitions.

Win32 is a subset of Windows 2000 (64-bit). So, include code guards for both in order to compile 32-bit code for the Intel IA-64 architecture.

Table 7 lists a few of the #ifdef code guards. The example shows you how you might use a code guard for the Intel IA-64 architecture. There are many code guards. Be sure you use the latest, standard, and up-to-date conditional compilation macros.

Table 7

Some New Code Guards

<code>__LP64__</code>	(On some UNIX* platforms) compilation is using the 64-bit UNIX data model, where <code>long</code> and <code>pointer</code> are 64 bits, and <code>int</code> remains 32 bits
<code>__M_IA64</code>	Compiler will generate code for IA-64 execution
<code>__unix</code>	Compilation is being done for running code on a UNIX platform
<code>__WIN64</code>	Compiler will generate code for Win64 API
<code>__WIN32</code>	Compiler will generate code for Win32 API (which will also run under Windows 2000, 64-bit)
<code>__WIN32_WINNT</code>	Compiler will generate code that runs only on the Windows NT* variant of Win32 (not Windows* 95 or 98)
<code>__WORDSIZE</code>	(On other UNIX platforms) compiler sets this macro to either 64 or 32

Example

Current Code IA-64 Code

<pre> #if defined (WIN32) ... #elif defined (__unix) ... #else #error unknown platform - revise _FILE_ _LINE_ #endif </pre>	<pre> #if defined (WIN32) or defined (WIN32) #if defined (__WIN64) //64-bit specific Windows code #else //32-bit specific Windows code #endif #elif defined (__unix) #if defined (__LP64__) (__WORDSIZE == 64) //UNIX/64 code #else //32-bit UNIX code #endif </pre>
---	--

Discussion

To define the first section of code for both Win32 and Windows 2000 (64-bit), the `__WIN64` conditional was added to the code. To include UNIX/64, `__LP64` was added to the code.

Remember

- If possible, use ANSI versions in your code, so it is compilable in any environment without a code guard.
- Be sure you use the latest, standard, and up-to-date conditional compilation macros.
- To include Win32 and Windows 2000 (64-bit) code in a code guard, add an appropriate definition for Windows 2000 (64-bit).

Big Integer Declarations

Since the largest integer type available differs between UNIX/64 and Windows 2000 (64-bit), `long` may not be used as the largest built-in integral type in both environments. Windows 2000 (64-bit) uses the `__int64` type. If the code is intended to be compiled for both UNIX/64 and Windows 2000 (64-bit), to make the code generic, you can add an `#ifdef` to test for the platform and use a `typedef` as shown in the following example.

Example

In-line #ifdef's Source Code

<pre> #ifdef WIN64 ... typedef __int64 int3264_t; #else ... #endif </pre>	<pre> long bigCount; Windows 2000 Post-compile Result int3264_t bigCount; // an int as big as the architecture permits </pre>
---	---

Windows* 2000 (64-bit) Window-Class Data

GUI-based applications create object “class” records to communicate information about how the Operating System sees and understands the application and its windows. Often it is convenient to combine storage of OS-related and application-related data in these same records.

In 32-bit Windows, the 32-bit data items could hold an integer, a pointer, or the parameters to the Win32 public functions `SetClassLong()`, `GetClassLong()`, `SetWindowLong()`, and `GetWindowLong()`. In Windows 2000 (64-bit), you must carefully determine the size of the stored data and calculate appropriate field offsets depending on the size of the data item – 32-bit or 64-bit. Then use new, updated public Set and Get functions to make these distinctions. These new functions are listed in Table 6.

Table 6

New Windows* 2000 (64-bit) Window Class Functions			
Scalable API	64-bit Compilation	32-bit Compilation	32-bit API
<code>GetClassLongPtr</code>	64-bit pointers	32-bit pointers	<code>GetClassLong</code>
<code>SetClassLongPtr</code>			<code>SetClassLong</code>
<code>GetWindowLongPtr</code>	64-bit offsets	32-bit offsets	<code>GetWindowLong</code>
<code>SetWindowLongPtr</code>			<code>SetWindowLong</code>

Example

Current Code

```
LONG lVal = GetWindowLong(hWnd, GWL_HINSTANCE);
LONG prev;
prev= SetWindowLong(hWnd, 12/*field 3*/, (long)pMyData);
```

IA-64 Code

```
LONG_PTR lVal = GetWindowLongPtr(hWnd, GWLP_HINSTANCE);
LONG_PTR prev;
prev= SetWindowLongPtr(hWnd, (int)(3*sizeof(LONG_PTR)), (LONG_PTR)pMyData); //C-style field
count 0, 1, 2, ...
```

Discussion

The IA-64 code makes use of the new scalable public functions, `GetWindowLongPtr` and `SetWindowLongPtr`. In case the get function receives a pointer-sized value, `lVal` was retyped to a scalable, pointer-precision type, `LONG_PTR`. `pMyData` was also changed to a pointer-precision type.

The `SetWindowLongPtr` function writes data that can be of a length depending on the architecture for which the code will be compiled. Since the code can be compiled to either IA-32 or IA-64 architecture, the memory address location to which it is written (field 3) will change depending on whether the data boundaries are 32 bits or 64 bits. To ensure the data is written to the correct location, the fixed-size offset of 12 in the Current Code is changed to automatically set the offset according to the architecture by using the `sizeof()` function and the scalar number 3.

Now the code will safely compile with types that scale with the architecture.

Remember

- Install the most recent Microsoft Platform SDK
- Use the *new* Window class functions.
- Check to make sure that any variable that is receiving or giving a value from or to a Window-class function is of a scalable type.
- Check all hard-coded offsets and change them to scale with the target architecture using the `sizeof()` function.
- This information applies to Windows 2000 (64-bit) only.

Using Formatted I/O Messages

IA-32 format specifiers used in formatted I/O messages, such as `printf`, `fprintf`, or `sprintf` may not be sufficient for referencing 64-bit quantities in the Intel IA-64 architecture.

To ensure your code uses large enough specifiers, use the directive `p` (ANSI C compatible), or use the Microsoft Visual C++*, `I64`. If you need to display a 64-bit integer in a formatted I/O message, prefix the integer with `p`, or use the size-specifying prefix `I64`, implemented in Microsoft's C runtime library.

The following examples show how you might use these for specifying formatted I/O messages.

Example

Current Code	IA-64 Code
<code>printf("pVal- %u\n", pVal);</code>	<code>printf("pVal- %p\n", pVal);</code>

Discussion

The Current Code prints the value of the pointer, `pVal` and assumes the value will fit into an unsigned-integer type. The ANSI formatted directive, `p`, is used in the IA-64 code to automatically scale with the architecture's size. This method works for both Windows and UNIX.

The Current Code prints the value pointed to by `pVal` assuming only a 32-bit wide value.

Example

Current Code	IA-64 Code
<pre>intptr_t *pVal; printf ("*pVal=0x%08lx\n", *pVal);</pre>	<pre>#if defined (WIN32) && defined (_WIN64) define FMTSZ3264 "I64" #else /*Win32 or UNIX */ define FMTSZ3264 "l" #endif ... intptr_t *pVal; printf ("*pVal=0x%0*" FMTSZ3264 "x\n", (int)(2*sizeof(*pVal)), *pVal);</pre>
Output: <code>*pVal= 0x054A7320</code>	Output: <code>*pVal= 0x00000003025A6000</code>

Discussion

The IA-64 Code uses code guards to define the format specifier `FMTSZ3264` as `I64` for Windows 2000 (64-bit) or `1` for Win32 or UNIX. The second `*` and the string `(int)(2*sizeof(*pVal))` make the number of printed digits to be correct for the architecture.

Remember

- Check all format specifiers to be sure they specify a large enough size.
- Use `p` if possible to ensure the code will compile across platforms.
- Use code guards, if necessary, to define format directives for either IA-32 or IA-64, and UNIX.

Section 3: Porting Pointers

Pointers differ in size in the Intel® IA-32 and Intel® IA-64 architectures. Pointers in the Intel IA-64 architecture are 64 bits. This section provides information on how this change might affect your code.

Type-casting Pointers

Pointers should be cast *only* to integral types that *scale with the architecture*, which means all instances of `(int)pointer` casts must be changed to scalable, or pointer-precision, types, such as `uintptr_t`. The pointers listed in **New Scalable Data Types in Section 2**, can be used for type-casting pointers. The following types are some examples of types that do *not* scale and should *never* be used to type-cast pointers.

- Windows* 2000 (64-bit): `LONG`, `ULONG`, `INT`, `UINT`, `DWORD`
- UNIX*: `int`, `unsigned int`

If you have code that interprets pointers as integral numeric data, such as for pointer arithmetic or bit manipulation of the pointer value, use the scalable, or pointer-precision, data types listed in **New Scalable Data Types in Section 2**.

If a pointer must be truncated to 32 bits, use caution and follow these guidelines:

- Windows 2000 (64-bit): Use `PtrToLong()` or `PtrToUlong()` to silence the pointer truncation warnings. A pointer that has passed through these functions should not be used as a pointer again.
- UNIX: You can do the same thing in UNIX with casting.

The following example shows how you might modify pointer-related code.

Example

Current Code

```
ImageBase=(PVOID)((ULONG)ImageBase | 1);
```

IA-64 Code

```
ImageBase=(PVOID)((ULONG_PTR)ImageBase | 1);
```

Discussion

This piece of code sets the `ImageBase` least significant bit; however, when ported to the IA64 architecture, using `ULONG` will truncate the pointer value, losing the upper 32 bits. By changing the data type to a pointer-precision type, `ULONG_PTR`, the size will scale with the architecture. The scalable data type, `uintptr_t`, could also be used, which would make this code easily portable to both UNIX/64 and Windows 2000 (64-bit) environments. In addition, using `void *` instead of `PVOID` would make this code more easily ported to both environments.

Pointers to Legacy Data

For Windows 2000 (64-bit) code that requires accessing legacy 32-bit data, you can use the following options:

- Use `#pragma pointer_size()` to specify which data structures are 32 bits and which ones are 64 bits.
- Use pointer modifiers to explicitly qualify a pointer, over-riding compiler flags or `#pragma pointer_size()` settings.

Pragmas and pointer modifiers enable calling 32-bit functions from 64-bit processes. Use pragma or pointer modifiers to access legacy data from any 32-bit source.

In UNIX, pointers are always 64-bits. [\[DOES THIS MEAN YOU CAN'T ACCESS 32-BIT DATA? WHAT IS THE IMPACT OF THIS STATEMENT? NEED HELP HERE.\]](#)

Example

Current Code	IA-64 Code
<pre> struct tags { int *a; int *b; int *c; }; struct data_record { struct tags t; struct data_record *next; }; struct data_record *foo; struct data_record nextfoo; ... foo->next=&nextfoo; </pre>	<pre> #pragma pack(push,1) #pragma pointer_size(push,32) struct tags { int *a; int *b; int *c; }; struct data_record { struct tags t; struct data_record *next; }; #pragma pointer_size(pop) #pragma pack(pop) struct data_record *foo; struct data_record nextfoo; ... foo->next=&nextfoo; </pre>

Discussion

If the Current Code is compiled for the Intel IA-64 architecture, the pointers a, b, and c would be 64 bits and would be truncated. In the IA-64 Code, the first set of pragmas sets the default pointer size to 32 bits, so the pointers to integers are 32 bits. The second set of pragmas reset the default pointer size to 64 bits. When the IA-64 Code is compiled for the IA-64 architecture, the pointers a, b, and c will be 32 bits long.

Accessing Data Structures With an Offset

In the Intel IA-64 architecture, data fields are naturally aligned, and the compiler might automatically pad data structure members for alignment reasons. To access the data structure using constant offsets, use one of the following macros:

- `offsetof()` macro defined in `<stddef.h>` (C/C++)
- `field_offset` macro defined in `<winnt.h>`

The following example shows how you might use the `offsetof()` macro.

Example

Current Code	IA-64 Code
<pre>#include <windows.h> #include <stdio.h> typedef struct _FOO { PCHAR pszName; DWORD dwVal; } FOO, *PFOO; int main(void) { PFOO pFoo = new FOO; ... DWORD dwVal= *(PDWORD) ((PBYTE)pFoo+4); printf("dwVal=%d\n", dwVal); delete pFoo; return 0; }</pre>	<pre>#include <windows.h> #include <stddef.h> #include <stdio.h> typedef struct _FOO { PCHAR pszName; DWORD dwVal; } FOO, *PFOO; int main(void) { PFOO pFoo = new FOO; ... DWORD dwVal= *(PDWORD) ((PBYTE)pFoo + offsetof(FOO, dwVal)); printf("dwVal=%d\n", dwVal); delete pFoo; return 0; }</pre>

Discussion

The Current Code contains a hard-coded offset to access a data item. When compiled for the IA-64 architecture, the hard-coded offset might not correctly identify the field. By changing the hard-coded offset to an offset that is automatically determined according to the architecture (32-bit or 64-bit) the field will be correctly identified in either architecture. The `<stddef.h>` file is included, because the `offsetof()` macro is defined in this file.

For more details on these topics, see the following sections:

- **Managing Data Size: Structure Padding in Section 6**
- **Managing Data Size: Pointer Cleaning in Section 6**

Remember

- Type-cast pointers only into scalable, or pointer-precision, types.
- Pointers are, by default, 64-bits. When accessing 32-bit structures, use a pragma or a pointer modifier.
- Hard-coded offsets to legacy data should be modified to use a macro that automatically determines the offset based on the architecture.

Section 4: Preparing for Multi-platform Compilation

Part of the porting process is to consider the organizational and build aspects of your files that make porting easier. This section covers the following cross-compilation topics:

- Compatibility files
- File naming
- Directories
- Build rules

Using a Compatibility File

You can increase the marketability of your software by making it available to as many platforms as possible. However, while both UNIX* and Windows* are based on ANSI C and C++, the UNIX and Windows worlds each include their own additions and differing extensions to these standards. The resulting differences make it appear difficult to write portable, multi-platform source code.

You can minimize this difficulty by defining a single set of types in a C header file (compatibility file) that you include in your source code. By including this header file and using conditional compilation, a single, uniform set of type names can be used across all platforms. For example, use of the `<portatyp.h>` file adds a small number of key-type names, which will aid in building multi-platform source code. By editing this file, you can establish Windows-like type names across platforms as well.

You can freely download `<portatyp.h>` from [\[URL HERE\]](#) and use it as a starting point, adapting it to the needs of your own projects. This file is not an Intel product, and [\[DISCLAIMERS HERE ABOUT THIS FILE\]](#).

File-Naming Conventions

Compiling for multiple platforms involves using many file naming conventions. Use the common file-naming conventions that are listed in Table 8 for source code and result files. These conventions will help you quickly identify the contents of the file and the target platform.

Table 8
File-Naming Conventions

	Object Files	Library Files	Exec. Files	Source File Names C++	Source File Names C	Notes
Windows	.obj	.dll, .lib	.exe	.cpp	.c	Not case sensitive
UNIX	.o	.so, .a		.C, .cpp, .cxx, .cc	.c	Case sensitive
Suggestions	Use a makefile variable			Use .cpp everywhere ⁶	Use .c everywhere ⁶	Preserve case

⁶ Differentiating Windows and UNIX source file extensions can reduce confusion when compiling for both platforms.

Directory Structure

If you are going to compile for more than one platform, you will need to update your directory structure. Create separate debugging and build directories for each platform on which you will compile your application. Figures 1 and 2 show directory examples.

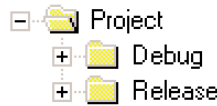


Figure 1. Single-target build directory

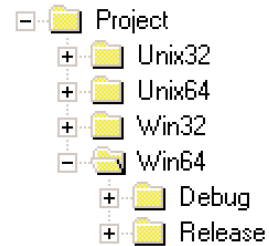


Figure 2. Multiple-target build directory

Build Rules

You may also need to update your build rules if you are going to compile for multiple platforms. If you use a cross-platform version of make or a similar build tool to build your software, you can write build rules usable on all platforms. Using such build rules requires compiling from a command line, but it does not prevent you from using visual environments, such as Microsoft Visual C++* or GNU* DDD*, to test and debug your program.

If your project is small, you might find it practical to create a separate platform-specific makefile for each platform. If your project is extensive, you might find it easier to create a main makefile with a tree of `if/else` rules through which you can access platform-specific `makedef` files. By setting the appropriate conditions, you can use one makefile for all platforms.

Most build tools provide a way to search along a path (a list of directories) for source files using directives, such as `.SOURCE` or `VPATH`. Careful placement of these directives in your platform-specific makefile allows the make tool to find platform-specific source code when necessary.

The following example might help you when creating a multi-platform makefile.

Example

UNIX-specific Makefile	Cross-platform Makefile
<pre>foo.o: foo.c abc.h ... foo: foo.o abc.a</pre>	<pre>foo\$(OBJ): foo\$(Csrc) abc\$(HDR) ... foo\$(EXE): foo\$(OBJ) abc\$(LIB)</pre>
Windows-specific Makefile	
<pre>foo.obj: foo.c abc.h ... foo.exe: foo.obj abc.lib</pre>	

Discussion

In the cross-platform makefile, the variables were defined in a platform-specific header file. When the line is expanded, the variable definitions will be correct for the appropriate platform.

Remember

- To make it easier to create files that can be easily compiled for multiple platforms, use a compatibility header file. Download the sample file and update it as necessary for your projects.
- Use naming conventions that make it easier to compile for multiple platforms.
- Compiling your application for multiple platforms may require you to update your directory structure and build rules.
- You can use an individual makefile for a small application that needs to be compiled for more than one platform.
- Create a cross-platform makefile with make-evaluated variables when compiling larger applications for multiple platforms.

Section 5: Calling Conventions

This section describes how the Intel® IA-64 architecture uses registers for function calls.

Function Call

The Intel IA-64 architecture uses multiple registers to pass arguments between caller and callee (Figure 3). Parameters are allocated onto a conceptual argument list, and then mapped to registers. When the registers fill, the remaining parameters are passed in the outgoing parameter area of the memory stack frame.

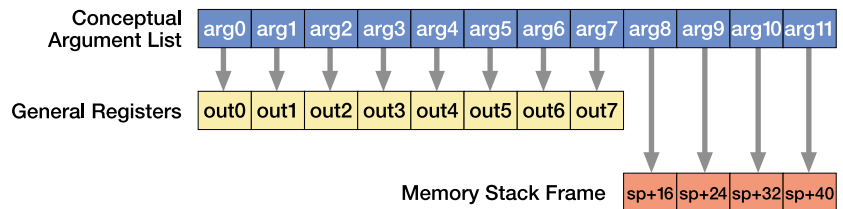


Figure 3. Function call concept

Argument slots can hold integral values, double-extended floating-point values, data structures, and arrays.

These rules also apply to functions with a variable number of parameters, such as printf.

For more information on calling conventions, see the **Intel IA-64 Software Conventions and Runtime Architecture Guide**, which you can download from the following site: <http://developer.intel.com/design/ia64/downloads/245256.htm>

Function Call Through General Registers

Function call return values of up to 256 bits are returned in registers according to the following rules (Figure 4):

- Integer values up to 64 bits are returned in general register r8; integer values up to 128 bits are returned in r8 and r9.
- Structures and arrays up to 256 bits are returned in the remaining general registers.
- AggregateType functions and other values greater than 256 bits are returned in a buffer allocated by the caller.

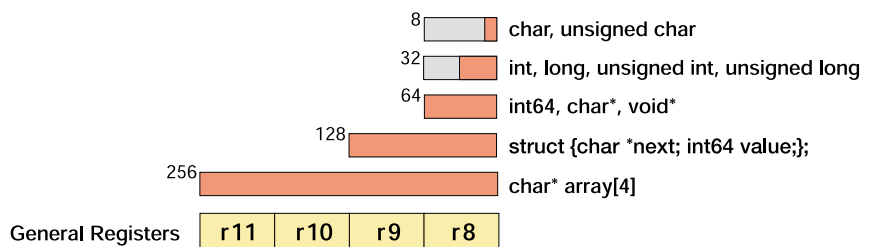


Figure 4. Function calls using general registers

The register stack engine clears the frame from the stack. For additional information, see the online tutorial **Introducing the IA-64 Architecture**, which is located at the following site: <http://developer.intel.com/vtune/cbts/ia64tuts/index.htm>

Function Return Through Floating-Point Registers

Floating-point values are returned according to the following rules (Figure 5):

- Single-, double-, and double-extended floating-point values are returned in f8.
- Quad-precision values are returned in the pair f8 and f9.
- Structures and arrays up to 8 elements are returned in the remaining floating-point registers.

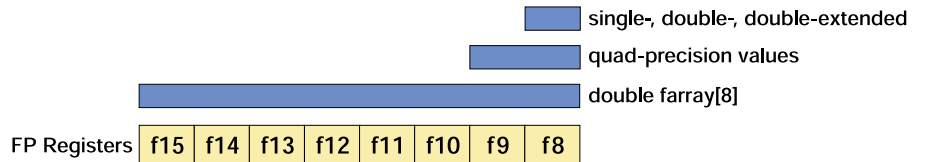


Figure 5. Function return through floating-point registers

Explicit Prototyping

If functions are not prototyped, ANSI C requires that parameters be passed as either `int` or `double`, which might be incorrect for your actual code logic. To avoid this problem, be sure to *explicitly* prototype all functions. This guideline applies to all programming platforms.

Explicit prototyping also provides these benefits:

- It ensures that the caller and the callee have matching argument types, because the compiler will evaluate the types and generate warnings for mismatched types.
- It protects against parameter misuse that can lead to parameter corruption.
- The compiler can more efficiently pass parameters to called functions.

Note that you should explicitly prototype a function in a header file to save time.

The following example illustrates prototyping.

Example

Without Explicit Prototype	With Explicit Prototype
<pre> 1 //module1.c 2 extern int initialize_buffer(); 3 4 void main(void) 5 { 6 char*p1=malloc(1000); 7 char*p2=malloc(2000); 8 int size=p2-p1; 9 initialize_buffer(size) 10 } 11 //module2.c 12 char * global_buffer; 13 int initialize_buffer(SIZE_T size) 14 { 15 global_buffer=(char*) malloc(size); 16 if (global_buffer==NULL) return(-1); 17 return(0); 18 } </pre>	<pre> 1 //module1.c 2 extern int initialize_buffer(SIZE_T); 3 4 void main(void) 5 { 6 char*p1=malloc(1000); 7 char*p2=malloc(2000); 8 SIZE_T size=p2-p1; 9 initialize_buffer(size) 10 } 11 //module2.c 12 char * global_buffer; 13 int initialize_buffer(SIZE_T size) 14 { 15 global_buffer=(char*) malloc(size); 16 if (global_buffer==NULL) return(-1); 17 return(0); 18 } </pre>

Discussion

At line 2, the prototype explicitly specifies the kind of parameters to be received by the function. In the Current Code, the size defaults to `int`, but in the Intel IA-64 code, it is a scalable type, `SIZE_T`, which ensures the code can be easily compiled for either 32-bit or 64-bit architectures.

At line 8, `size` is specified as an `int` in the code Without Explicit Prototype. The `int` declaration limits the difference between two pointers to no greater than 2^{32} . This limit becomes a problem when compiled for the IA-64 architecture, because values greater than the limit will be truncated when passed to `size`. In the code With Explicit Prototype, a scalable type, `SIZE_T`, is used to ensure that the size of the difference between the pointers adapts to the architecture. When compiled for the IA-64 architecture, the difference between pointers can be up to 2^{64} , ensuring that the upper 32 bits won't be lost.

At line 13, the callee expects a `SIZE_T` parameter, which, when compiled for Windows* 2000 (64-bits) is a 64-bit value. But the `int` in the code Without Explicit Prototype supplies a value only for the low order 32 bits; the upper 32 bits contain unknown information. In the code With Explicit Prototype, the callee expects and receives a 64-bit value. When compiled for the Intel® IA-32 architecture, `SIZE_T` will ensure that the callee still receives the proper size of value.

Explicit Prototyping of Floating-Point Functions

When an explicit prototype exists for a floating-point function, the floating-point values are passed in the next available floating-point register (Figure 6). Omitting the explicit prototype causes the following:

- Floating-point parameters consume duplicate resources, because they are copied to both general and floating-point registers, resulting in degraded performance.
- Parameters are corrupted, because unknown data can end up in registers.

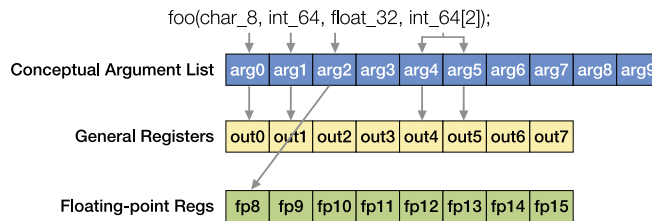


Figure 6. Passing data through floating-point registers

The following examples illustrate duplicate resources and parameter corruption.

Example Consuming Duplicate Resources

Without Explicit Prototype	With Explicit Prototype
<pre>extern int func(); VOID main (VOID) { int i = 1; int j = 2; double a = 2.333; double b = 256.6666; func(i, a, b, j); ... }</pre>	<pre>extern int func(int, double, double, int); VOID main (VOID) { int i = 1; int j = 2; double a = 2.333; double b = 256.6666; func(i, a, b, j); ... }</pre>

Discussion

Without explicit prototypes, all variables are first copied to general registers, and the floating-point values are also copied to the floating-point registers. This consumes two extra registers and leaves a and b needlessly duplicated in the general registers (Figure 7).

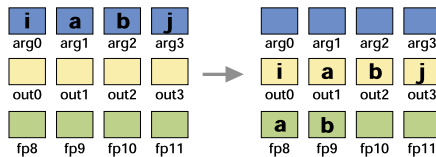


Figure 7. Passing data to registers without explicit floating-point prototyping

When the function is explicitly prototyped, the int values are copied to general registers, and the floating-point values are copied only to floating-point registers (Figure 8).

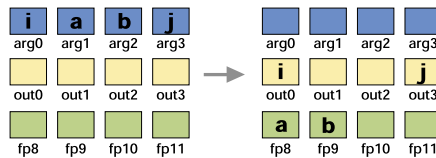


Figure 8. Passing data to registers with explicit floating-point prototyping

Example Consuming Duplicate Resources

Without Explicit Prototype	With Explicit Prototype
<pre>file1.c: void fun(float f8){ printf("%f/n", v); } file2.c: main() { fun(1); }</pre>	<pre>file1.c: void fun(float f8){ printf("%f/n", v); } file2.c: extern void fun (float f8); main() { fun(1); }</pre>

Discussion

fun is defined in file1.c but called in main, which is in file2.c. Without an explicit prototype in file2.c, main doesn't see the fun prototype. main doesn't know that fun should return a floating-point value. When fun is called, without an explicit prototype, it passes an integer value, which is stored in a general register, and the floating-point register contains unknown information. When the function is properly prototyped, as in the code With Explicit Prototype, it passes a floating-point value, which is stored in a floating-point register. With an explicit prototype, the function expects and receives a floating-point value.

Remember

- In function calls, parameters are mapped to a conceptual argument list and then to registers.
- Depending on the size and type of the value, a parameter can occupy one or more general or floating-point registers, or a buffer allocated by the caller.
- Always provide explicit prototypes of functions to ensure the caller and callee are the same type. Otherwise, the compiler will automatically path to a type that is possibly not appropriate for your function.
- Always provide an explicit prototype for floating-point functions to eliminate duplicating resources and corrupting parameters.

Section 6: Manual Code Cleaning

Some aspects of code cleaning are not as obvious as others. They take a little more analysis of the code beyond the issues discussed so far. These aspects include managing data size and hash algorithms, which are discussed in this section.

Managing Data Size

General Rule

To conserve resources and avoid data bloat, be sure you are using data types that are appropriate for your variables; consider how large your variables must count. If a variable won't need, for the foreseeable future, to count past 2^{32} (4,294,967,296), it may be better to NOT make it a 64-bit data type.

If you know that the variable does not need to be pointer polymorphic (scale with the architecture), use the following guideline to see if it can be typed as 32-bit instead of 64-bit. (This guideline is based on a data expansion model of 1.5 bits per year over 10 years.)

1. Consider your largest data range, for example 0 ... 1,000,000
2. Multiply the range by 1.5^{10} (approximately 58)
3. If the result is still less than 2^{32} , code the variable as a 32-bit variable, such as `int`.
4. If the result is larger than 2^{32} , or you know of impending need for capacity increases, code the variable as a 64-bit variable, such as `long` (UNIX*/64 only) or `__int64` (Windows* 2000, 64-bit only).

Structure Padding

With the Intel® IA-64 architecture, data boundaries are naturally aligned, instead of freely (any byte) aligned on IA-32 architectures. Depending on the field order in a 64-bit `struct`, this change in boundaries may lead to padding of 32-bit fields, causing data bloat (Figure 9). Padding fields, especially if there are many `structs` called with both 32-bit and 64-bit fields, could significantly degrade the performance of your application.

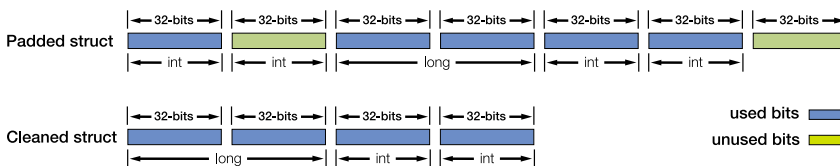


Figure 9. Field boundaries with IA-64 architecture

To reduce or eliminate padding, reorder your field declarations in `structs` so that the 64-bit fields are first, followed by 32-bit fields (or vice versa, although this could end up in padding the last field). If the record is shared with non-IA-64 computers, you will need to write the `struct` accordingly.

Example

Current (Win32) Code	Revised (Windows* 2000) Code
<pre>struct dim_t { int height; long width; int weight; };</pre>	<pre>struct dim_t { __int3264 width; int height; int weight; };</pre>

Discussion

The following example shows how you might rearrange the order of fields. If the Current Code is compiled for the IA-64 architecture, the two variables, height and weight, would end up being padded, because they are 32-bit variables bounded by 64-bit boundaries. This creates data bloat and reduces application performance. By rearranging the field declarations and using a scalable Windows 2000 (64-bit) data type, padding (and data bloat) is eliminated, while making the code compilable for IA-32 or IA-64 bit architectures.

Pointer Cleaning

Pointers in the Intel IA-64 architecture are twice the size of pointers in IA-32, which may effectively double the size of data structures largely composed of pointers. You can redesign data structures that contain lots of pointers to use 32-bit offsets from a base pointer instead of using full 64-bit pointers; this will reduce bloat. To do this, you can use a Windows-compatible compiler and apply the `__based` attribute to pointers. You might also be able to convert pointers into smaller-sized array indexes. Modifications like these must be accompanied by changes to your application source code. Instead of a simple C de-reference (`->`), your code must calculate an address, or you must change it to use array subscripting (`[]`). Careful use of macros or C++ overloaded operators can preserve easy readability of your source code.

Example

Current Code	Revised Code
<pre>struct f { f *Pnextf; g *Psibling; h *Pparent; };</pre>	<pre>1 typedef HALF_PTR int; 2 void *heap; 3 struct f { 4 int nextf_idx; 5 HALF_PTR siblingOffset; 6 h __based(heap) *_ptr32 Pparent 7 }; 8 ... 9 f_array[nextf_idx] 10 (g*)((uintptr_t)this+this.siblingOffset 11 this->Pparent</pre>

Discussion

The Current Code defines a structure composed entirely of pointers; however, the Revised Code defines a structure (`struct f`) composed of smaller data fields that will be used to point to elements of the structure. Line 2 defines a pointer to heap that allocates objects within a 32-bit address sub-range. Line 6 uses a Microsoft*-specific extension (`__based`) to create an offset within the heap sufficient to reach all allocated members. Line 5 declares a data type that will be used as an offset to a pointer. Lines 9 through 11 declare that the data item is pointed to by a pointer-plus-offset calculation instead of direct 64-bit addressing. Line 9 defines a simple-array index. Line 10 is a base + displacement calculation that needs to be inserted into your code. Line 11: The complicated declaration in the structure (line 6) allows simple, traditional, straightforward use of the structure fields in user source. Revising your code to use this construction will allow you to use these smaller “pointer-like” fields.

Remember

To effectively manage data size,

- Don't use a larger data type than you need for non-pointer-polymorphic variables. Use the general rule to determine whether to use a 32-bit or 64-bit type.
- To reduce data bloat, check your data structures to see if you need to rearrange field declarations.
- Pointers are 64-bits. You may need to modify your code to use 32-bit offsets from a base pointer or use an array.
- You may need to modify algorithms to properly calculate offsets.

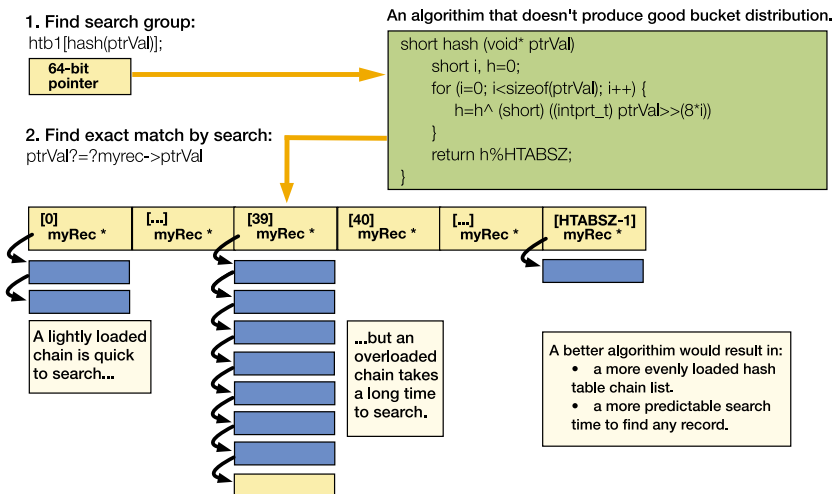
Hash Algorithms

An algorithm that works in IA-32 may not work as well in IA-64. A poorly chosen algorithm may result in an unbalanced hash table in which significant data is masked out. You should check shifting and logical operations in your algorithm.

When using pointers and seek keys as hash keys, be aware that 32-bit pointers and object identifiers have approximately 30 significant, varying-address bits, while 64-bit pointers or object identifiers have about 35 significant, varying-address bits, as well as many zeroes and ones that are largely unchanging.

Ensure that your algorithm scrambles the appropriate data values and that it can log and report hash chain length. Operate on significant bits of pointers.

The following example might help you find areas in your hash algorithms that need to be modified.



Section 7: Miscellaneous Topics

Jump Buffer

When using `set jmp` and `long jmp`, use the ANSI-defined `jmp_buf` data type. `set jmp` saves context into the `jmp_buf` buffer, which is used later by `long jmp`.

When using a non-ANSI-defined `jmp_buf` data type, code might allocate its own jump buffer, which might have a size valid for existing Intel® IA-32 code, but not for Intel® IA-64 code. You therefore need to examine the jump buffer size to ensure that it is adequate. The following example shows how you might modify code that uses jump buffers.

Example

Current Code	Revised Code
<code>DWORD my_jmpbuf[16]</code>	<code>jmp_buf my_jmpbuf;</code>

Discussion

In the Current Code, enough space is allocated in the buffer for 16 IA-32 pointers, but only 8 IA-64 pointers. By using `jmp_buf`, the code ensures that enough space is allocated for the jump buffer, for both architectures. As with offsets, it is best not to hard-code data when a system-defined feature can be used instead.

Thread Stacks

Both Windows* and UNIX* operating systems permit multiple threads of execution within a process' address space. Each thread is given its own execution context, including its own call/return stack. These stacks must be of sufficient size to store all of the likely call chains the thread might experience while it is running.

Because 64-bit applications will, in general, pass larger-sized items in function parameters, you must re-assess prior decisions or defaults for stack and guard region sizes. Both operating systems implement:

- A way to specify the initial size for a thread's stack.
- A “guard” region at the “top” of the allocated stack space that will cause a runtime fault if it is accessed.

The guard region is a way to inform you that the thread's stack has been exceeded and to prevent unobserved damage to data owned by other threads or data structures in the process.

UNIX Thread Stack

The POSIX* threads facility, which is implemented by current UNIX operating systems, provides each created thread with a call/return stack. The stack size is defined by the specific implementation of UNIX. If you allow the OS to create and manage the thread's stack for you, a “stack guard” region is also allocated automatically.

You can extract the default sizes for the stack and guard regions by compiling and running the following piece of code:

```
pthread_attr_t thrAttr;
size_t sizeStack, sizeGuard;
pthread_attr_init(&thrAttr);
pthread_attr_getstacksize(&thrAttr, &sizeStack);
pthread_attr_getguardsize(&thrAttr, &sizeGuard);
```

If the values that are returned are not appropriate for your application, you must adjust the following configuration settings prior to creating a thread. The following piece of code illustrates how to modify the configuration settings:

```
pthread_t thrHandle;
pthread_attr_setstacksize(&thrAttr, 1000000); // a megabyte
pthread_attr_setguardsize(&thrAttr, PAGESIZE*8); // 8 OS pages
pthread_create(&thrHandle, & /*adjusted*/thrAttr, ...);
```

Windows Thread Stack

The existing thread stack size may be inadequate for the Intel IA-64 architecture. To automatically allocate sufficient stack space in Windows NT*, use stack probes: /Ge for global checking, or #pragma check_stack(on) for checking specific functions. If you decide to increase performance by disabling stack probing, look for thread creation and manually check the stack sizes to ensure that they are large enough.

The following code example shows how you might allocate enough thread stack space while monitoring functions using stack probing.

Example

Win32 Code Without Stack Probe	Windows* 2000 With Stack
<pre>int g_cnt=0; #define MAX_ITERATIONS 500 #pragma check_stack(off) DWORD ThreadFunc(LPDWORD param) { int i[1024]; i[0]=0; g_cnt++; if (g_cnt < MAX_ITERATIONS) ThreadFunc(NULL); return 0; } #pragma check_stack(on) ... HANDLE hThread=CreateThread (0, 1000*1024, (LPTHREAD_START_ROUTINE) ThreadFunc, &param, 0, &id;</pre>	<pre>1 int g_cnt=0; 2 #define MAX_ITERATIONS 500 3 #pragma check_stack(on) 4 DWORD ThreadFunc(LPDWORD param) 5 { 6 int i[1024]; 7 i[0]=0; 8 g_cnt++; 9 if (g_cnt < MAX_ITERATIONS) 10 ThreadFunc(NULL); 11 return 0; 12 } 13 ... 14 HANDLE hThread=CreateThread 15 (0, 1000*1024, 16 (LPTHREAD_START_ROUTINE) 17 ThreadFunc, &param, 0, &id; 18</pre>

Discussion

The code sets a counter to check how many times ThreadFunc executes. The function will iterate no more than 500 times. Normally, stack probes are automatically turned on by line 3 (#pragma check_stack(on) is inserted for clarity); however, the Win32 code turns the stack probes off. The code then defines and initializes the array that allocates the stack space. The counter is incremented, and the array is recursively called the number of times specified in MAX_ITERATIONS. At line 13 the stack probes are normally turned on automatically; the command in the Win32 code to turn on stack probes was added here for clarity.

In the last few lines of code, 1024 bytes of stack are allocated. ThreadFunc uses 4096 bytes every time it is called. In the Win32 code, by the 250th iteration, the allocated stack space of 1MB is used up, and the 251st iteration causes a stack overflow. In the Windows 2000 code, stack probing automatically allocates stack space for each iteration after the 250th.

Context Structure

A signal CONTEXT structure contains processor-specific register data and other machine state.

UNIX Context

UNIX saves machine state in context structures whenever an asynchronous transfer of control occurs, such as by a signal or signals.

The <signal.h> header file provides definitions of the structure.

The following example shows **[NEED DESCRIPTION HERE]**
THE COMMENT IN THE #ELSE LOOKS WRONG. NEED HELP HERE TO UNDERSTAND THE EXAMPLE.

Example

Current Code

```
#include signal.h
/* struct sigcontext thread_context == context structure
sent into handler function */
printf("Dump of Integer Context\n");
#ifdef __ia64__
printf("r15: %lx, r14:%lx\n", thread_context.sc_gr[r15],
thread_context.sc_gr[r14]);
#else /*__ia64__ */
printf("Edi: %08x, Esi: %08x\n", thread_context.edi,
thread_context.esi);
#endif
```

Windows Context

Windows* 2000 (64-bit) uses CONTEXT structures to perform various internal operations. If your application accesses the CONTEXT structure, you need to change the access to the Windows 2000 (64-bit) CONTEXT.

The <winnt.h> header file provides definitions of these structures.

The following example shows how you might combine Win32 and Windows 2000 (64-bit) code to maintain a single-code base that will compile for both architectures.

Example

Current Code

```
CONTEXT ThreadContext;
HANDLE hThread;
hThread = CreateThread(...);

SuspendThread(hThread);
ThreadContext.ContextFlags=CONTEXT_INTEGER;
GetThreadContext(hThread, &ThreadContext);
printf("Dump of Integer Context\n");
#ifdef _M_IA64
printf("r2:0%.16I64x, r3:%0.16I64x\n", ThreadContext.IntT0,
ThreadContext.IntT1);
#elif defined(_M_IX86)
printf("Edi:%08x, Esi: %08x\n", ThreadContext.Edi, ThreadContext.Esi);
#endif
```

Constants

When `#define` is used for a constant, the compiler cannot check the code for type mismatches. To let the compiler check for type mismatches, always specify a data type and use the ANSI C `const` to declare constants. During compilation, if there are any type mismatches between the declaration and implementation, the compiler will warn you.

The following example shows how you can use the ANSI `const` instead of `#define`.

Example

Defining a Constant with `#define` Defining a Constant with `const`

<code>#define mask 0x37FFC;</code>	<code>const int mask = 0x37FFC;</code>
------------------------------------	--

Discussion

Using `const` instead of `#define` allows the compiler to check the declaration for you.

Integer-Constant-Type Suffixes

If integer-constant-type suffixes are used in the code, you might need to modify the code, because the `L` or `l` suffix (meaning `long`) in Windows 2000 (64-bit) is a 32-bit type. Windows 2000 (64-bit) uses the `i64` suffix.

If the code is intended to be compiled for both UNIX/64 and Windows 2000 (64-bit), to make the code generic, you can add a centrally defined macro in the `<portatyp.h>` file as shown in the following example:

Example

portatyp.h

Source Code

<pre>#ifndef _WIN64 #define CONST3264(a) (a##i64) #else #define CONST3264(a) (a##L) #endif</pre>	<pre>// from UNIX/64-cleaned code const long ll= 3015L;</pre>
	<p>Windows 2000 Post-compile Result</p> <pre>const long ll= CONST3264(3015);</pre>

Discussion

To ensure compilation across platforms, the `<portatyp.h>` file includes type definitions that allow the use of a common type (`CONST364`) that will compile using the `i64` suffix for Windows 2000 (64-bit) and the `L` suffix for UNIX/64.

Hex Constants

If you used hex constants to generate a particular value, you might need to modify your code, because the value might change when porting to the 64-bit architecture. In addition, if you use a `#define` to define the constant, the compiler cannot check the type where the constant is used. You can end up with an inappropriate type for the value in your code.

For example,

```
0xFFFFFFFF
```

in IA-32 is `-1`, but in IA-64 is `4,294,967,295`; and

```
0x100000000
```

in IA-32 is `0`, but in IA-64 is `4,294,967,296`

Evaluate your code for hex constants and replace them with a C `const` declaration, such as,

```
const int allIs= 0xFFFFFFFF;
```

In addition, be sure to specify a particular data type, including whether it is signed or unsigned, and use type suffixes as appropriate, such as `L` or `UL`.

Truncation of long via doubles

The use of `long` and `double` is no longer compatible in the Intel IA-64 architecture. A `double` has 52 significant bits, and, in the 32-bit architecture, `double` can hold all the significant bits. But in the IA-64 architecture, the precision of `double` is smaller than a 64-bit integer, and bits can be lost.

If you use `long` and `double`, you should verify each instance to make sure it is acceptable, especially if significant digits will be lost when compiled for the IA-64 architecture. Use `long double`, if possible.

The following example shows how `long` and `double` can cause problems in your code:

Example

Current Code

```
long l; // if UNIX
__int64 l; // if Windows

double d;
extern double f(double);
...
d= l; // loses significant bits
l= d; // this can be a problem, also
...
d= f(l);
```

Section 8: Checklist

Use this list to help you organize your code preparation tasks. This list is meant to be an aid. There may be additional tasks required for your project that are not listed here.

Check for:	In this guide, see section
<input type="checkbox"/> Invalid cast between pointers and ints and/or long	2, 3
<input type="checkbox"/> Invalid printf specifiers	2
<input type="checkbox"/> Usage of existing APIs with pointer and/or long as parameters and/or return values	2
<input type="checkbox"/> Usage of undocumented and/or reserved-bit fields	Not covered
<input type="checkbox"/> Hard-coded values for sizes of data types	Not covered
<input type="checkbox"/> Hard-coded values for bit-shift values	Not covered
<input type="checkbox"/> Hard-coded constants in memory allocation functions	5
<input type="checkbox"/> Unguarded <code>ifdefs</code> from defaulting to unwanted code generation	2
<input type="checkbox"/> Accessing data structure members via constant offsets	3
<input type="checkbox"/> Algorithms that make use of special bits in pointer arithmetic assuming fixed size	3
<input type="checkbox"/> In-line assembly code	1
<input type="checkbox"/> Self-modifying code	Not covered
<input type="checkbox"/> Appropriately modify portions of code storing/restoring on-disk structures	Not covered
<input type="checkbox"/> Portions of code utilizing data-packing	Not covered

Section 9: Additional Resources

Intel is dedicated to providing as much information as possible to help developers in their efforts of preparing code for the Intel® IA-64 architecture. Besides this guide, there are other sources of information available from Intel and other OSVs, ISVs, and organizations. Some of these sources are listed below.

IA-64 Application Developer's Guide

Available from Intel. Use Intel literature order number #245188-001

IA-64 Software Conventions and Runtime Architecture Guide

Available from Intel. Use Intel literature order number #245256-001

IA-64 Tutorials

<http://developer.intel.com/vtune/cbts/ia64tuts/index.htm>

IA-64 Developer Web Site

<http://developer.intel.com/design/ia64/index.htm>

UNIX* I32, ILP64 Rationale

http://www.opengroup.org/public/tech/aspens/lp64_wp.htm

Windows* 2000 (64-bit) Programming Information

http://msdn.microsoft.com/library/sdkdoc/buildapp/64bitwin_410z.htm

Section 10: Intel® IA-64 Architecture Features

There are features of the Intel® IA-64 architecture that are beyond the current Win32 model. Making use of these features means better performance from your application; however, it may also require additional programming effort.

A significant IA-64 architecture feature is that it provides a minimum of 512GB of user memory. Redesigning your application to use the additional memory allows larger data sets to reside in memory, which results in better performance. For example, a Win32 database application that builds indexes for data searches may not be able to build a complete index of data due to memory constraints.

In addition, Intel and Microsoft* are developing pragmas and intrinsics that will enable making use of other IA-64 architecture features, such as predication and prediction strategy hints.

For more information on the Intel IA-64 architecture, see **Introducing the IA-64 Architecture** tutorial at the following site:

<http://developer.intel.com/vtune/cbts/ia64tuts/index.htm>

Glossary

This glossary lists some important terms you might want to know about as you prepare your code for the Intel® IA-64 architecture. You may or may not be familiar with these terms.

Base Type

A language-defined, built-in data type like long or short.

Cardinality

The range of numbers a data item can count.

Code Clean

Revising source code to be compilable in both 64-bit and 32-bit environments.

Data Bloat

Increase in size of data (especially structures) in 64-bit applications.

Derived Type

A type defined by an OS or function library to more clearly indicate a type's purpose or size, like `HANDLE` or `time_t`. These are defined in terms of a base type using `typedef` or `#define`.

Natural Alignment

Data is "naturally aligned" when it is positioned in memory at an address that is a multiple of its size (in bytes). For example, an `int` (size of 4 bytes) is naturally aligned at addresses 0, 4, 8, ..., (i.e., the 2 low-order bits of the address are 0).

Polymorphic

One data item having a different type to different users/viewers. Polymorphism is a key source of code-clean problems.

Reserved Word

A source code identifier with a special meaning to the OS, compiler, source code language, header files, or function call libraries. New ones have appeared; you must not redefine them.



Information in this document is provided in connection with Intel products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice. For most current product information, please visit <http://www.intel.com/network>

© 2000 Intel Corporation. All rights reserved.

* Third-party brands and names are the property of their respective owners.

0200/OC/RK/EW/PT/xx NPxxxx