Porting to Linux* for the Intel® Itanium™ Architecture

Information for Developers and ISVs

Martha Maroney
Trimar Communications

October, 2001

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.

Copyright © Intel Corporation 2001
* Other names and brands may be claimed as the property of others.
Contents

INTRODUCTION................................................................................................................... 1

OVERVIEW OF LINUX FOR THE INTEL® ITANIUM™ ARCHITECTURE .............1

SOFTWARE PORTING METHODOLOGY .................................................................2

   STEP ONE: UNCOVERING THE DEPENDENCIES .................................................. 2
   STEP TWO: PORT TO LINUX INTEL® ITANIUM™ ARCHITECTURE.................. 3
   STEP 3: CODE CLEAN CHECKLIST........................................................................ 3
      Following GNU development guidelines.............................................................. 4
   STEP 4: SELECTING DEVELOPMENT TOOLS AND TOOLKITS FOR LINUX...... 11
   STEP 5: Compiling the Application on Linux for the Intel® Itanium™ Architecture ... 12

FOR MORE INFORMATION . . .................................................................................. 13
Introduction

In recent years, we have seen phenomenal growth in use of Linux* operating systems on Intel® Architecture platforms. With its ties to the Unix operating system, Linux has become an operating system that represents speed, efficiency, flexibility, and stability. Because of these advantages, more and more companies are considering deployment of the Linux operating system in their systems that employ Intel® Architecture: from implementing it on desktops and workstations, all the way to front-end server and mission critical enterprise servers that run e-Business data centers. This represents a great opportunity for software companies and individual contributors working in open-source environments. With the availability of the Intel® Itanium™ processor, Linux promises to become a credible force in the enterprise computing market.

The benefits of porting to Linux for the Intel® Itanium™ Architecture (formerly referred to as Linux for IA-64) are significant. As the Itanium™- based platform gains popularity, it will make good economic sense for companies to retain and leverage a proven source of value and tap the ever-growing new market segment. The benefit does not simply stop there, however. Implementing Linux on the Intel® Itanium™ processor family can enhance the usefulness of existing applications with the added performance, scalability, and reliability of the Intel® Itanium™ processor.

This document is meant to address issues concerning porting existing applications on other platforms to Linux for the Intel® Itanium™ Architecture as well as call out the key issues for developing new applications on an Itanium™- based platform.

A substantial part of this paper offers background information on the subject of software portability. Because much current software was created in an environment where the time to market was the primary concern, portability of the software assumed lesser importance, and was the first thing to go when project deadline was at risk. Therefore this paper will also discuss more recent software design technologies to help software developers effectively address portability and enable them to design software that is easy to port and easy to maintain.

Overview of Linux for the Intel® Itanium™ Architecture

Porting Linux applications for use on Itanium™- based systems is a relatively easy and straightforward process, for several reasons. First, the designers of Linux for the Intel® Itanium™ Architecture have made a great effort to ensure the Linux APIs are clean. For example, the Itanium™- based systems version of Linux is a pure 64-bit operating system in that it features both 64-bit integer size and 64-bit pointer size. Compared to other quasi 64-bit programming models, a pure 64-bit model has the advantage of being an easy and straightforward software development and porting platform. Second, the source-level APIs are designed to be compatible with Linux for Itanium™- based systems wherever possible. For example, the error codes, signal codes and ioctl- codes have kept their own values. Third, there is a wide range of choices of software development tools for all stages of software development cycles.

Most of the time, these tools are also available on other platforms. Using the same tools across platforms usually results in fewer porting problems and higher software qualities. Fourth, These tools are designed
in a way to allow developers to migrate their software from other platforms. For example, Red Hat’s GNUPro®, which is one of the most popular software toolsets for Linux, has special features to help developers port code between big and little endians, across different integer and pointer sizes, and to systems with different alignment rules. The GNU C runtime library that comes with standard Linux releases captures the platform specific system calls in a form of standard library function calls.

In terms of standards compliance, Linux supports ISO and ANSI C and C++. Also, Linux incorporates and extends to some extent the Unix 98 APIs. All of which should help to minimize the problems you encounter when moving your Linux-based applications to the Itanium™-based platform.

Software Porting Methodology
The porting of software to Linux for the Intel® Itanium™ Architecture™ can be broken into five major steps as outlined in the sections below.

Step One: Uncovering the Dependencies

Before you start a porting project, you need to answer this obvious question: How portable is this software? What are the major porting hazards? Where are they? To answer these questions we need to identify the software dependencies, which can be divided into internal and external components.

The internal dependencies are the dependencies of the software source code on the development tools. To uncover the internal dependencies, you need to identify what compilers are used to build the application on the current host and on the target host. You also need to find out how similar they are and to what extent they conform to ANSI/ISO standard.

Each compiler has language extensions that may or may not be part of standard. A good practice in general is to stay away from those extensions. If you have to use those language extensions, you should find out and resolve the syntax and semantics differences, if there are any, in the compilers you will to use. For example, you can locate GNU C language extensions at [http://gcc.gnu.org](http://gcc.gnu.org). To minimize the work related to language extensions, a good practice is to stay with one compiler as much as possible while porting the application across the platforms to Linux for the Intel® Itanium™ Architecture™. At least three compilers, GNUPro C/C++, Intel C/C++ and Silicon Graphics C/C++, are available for multiple platforms. In general, the more strictly the compiler conforms to the standards, the more portable the code is; i.e. the less your code uses language extensions, the more portable your code will be. It is more difficult to port an application built by a non-ANSI compatible compiler to a new platform using only an ANSI compiler.

Understanding the internal dependencies of an application provides a rough assessment about the amount of the language related effort. Cleaning up those language related changes is a good investment either for future porting projects or for source maintainability. We will revisit the code clean subject in Step 3.

The external dependencies are the dependencies of the software on library or APIs provided by 3rd parties. Examples include BSD system calls, pthreads library, Microsoft MFC®, and Windows® SDK. To some extent, the portability of your application is determined by the portability of these library APIs used in your application. In step 3, Code Clean, we will address GUI library issues in more detail.
Intel has made an effort to help the software companies that provide those library APIs to port their products, and to extend their APIs to Linux for the Intel® Itanium™ Architecture. Tools such as Perl, tcl/tk, Rogue Wave include libraries, and the SmartHeap runtime library are already available on Linux for the Intel® Itanium™ Architecture. To get a comprehensive list of 3rd party software tools, please visit http://developer.intel.com/.

**Step Two: Port to Linux Intel® Itanium™ Architecture™**

After you have identified internal and external dependencies, you may consider rebuilding your applications for Linux on an IA-32 platform. This allows you to clean up your code logically in the Linux environment, eliminating errors and portability warnings, before you attempt it on the Itanium™ -based platform. Additional advantages are that porting to the familiar IA-32 Linux platform can enable you to quickly jump-start your project, to achieve a single source code on both IA-32 and Itanium™-based platforms, and to take advantage of the many cross-platform development tools. For example, lint64, a type-checking utility used for diagnosis on your application, is available on Linux for IA-32 releases.

While you are building your application on Linux for an IA-32 system, you may want to go over all the warning messages to investigate if they reveal any portability related issues. Rewriting a portion of the code that does not conform to standards or has high portability hazards is usually a good investment with high payoffs later on native platforms. In the end, you should be able to build a high quality, 32-bit Linux application on an IA-32 platform.

If you are porting from a Windows application to Linux for Itanium™-based systems, this step will be even more valuable. In addition, you should set a clear expectation as to which functionality you want to port yourself, and which functionality you can get by choosing a good GUI package. You may also want to rewrite some of the system calls in the C Runtime library calls or a POSIX-like function rather than Microsoft specific interfaces. You may also want to consider using 3rd party runtime libraries that allow you to maintain the existing GUI interface and still be able to run on the new platforms. Examples of such libraries are cygwin.dll, or MainWin.

**Step 3: Code Clean Checklist**

Code Clean is the process to update your source code with the standard data types and operations, standard runtime library calls, and any operating system’s multi-platform API. You can also extend the Code Clean process to include activities to restructure your application to hide the platform specific details in a few easily identifiable modules. The Code Clean step is absolutely critical, and the benefits are numerous. One often-overlooked benefit is its ability to add reliability, robustness, and maintainability to your software.

In this section, we will start with the process guidelines discussion, followed by design principles, and a checklist of porting hazards.
Following GNU development guidelines

One reason legacy software often is not portable is that the developer did not follow any guidelines to help ensure portability. GNU software, with more than 15 years of open, cooperative software development has provided a good example of portable software. As a result, it gradually evolved a set of conventions and guidelines loosely called the “GNU Guidelines”. GNUPro Software Development tools, which includes, gcc, g++, gas, and gld on Linux for the Intel® Itanium™ Architecture, best exemplifies portable software produced under such a guideline.

Two things stands out from GNU software guidelines

Use ISO/ANSI C/C++

If you use the GNU compiler, you should use the -ANSI option, which will restrict you to the most widely accepted interpretation of ANSI standards. Part of the standard defines the proper data types you should use: For example, you should use:

- `int32_t`
- `int64_t`
- `u_int8_t`

rather than

- `int`
- `long ( or long long)`
- `unsigned char`

Use configuration and platform support packages that use autoconf, automake, and libtool. Besides making it easier to maintain a single source base for virtually any platforms, including IA-32 and Itanium™ - bases platforms, these scripts capture the development environment differences such as the name and location of the tools, the name of the system calls, and the name and location runtime libraries. The scripts produce include files for your source code and a makefile, at which point you can start your building process with minimum human intervention.

Understanding the Data Model, alignment requirement, and Endianess

Linux for the Intel® Itanium™ Architecture uses the data model called LP64 only. Here are basic facts about this data model:
The best way to remember the facts about LP64 data model is to remember the 64-bit data types. They are
long int and pointer type. In the 32-bit world, Linux for IA-32 uses a data model called IPL32, in which
int, long int and pointer types are all 32-bit. On Linux for the Intel® Itanium™ Architecture, ILP32 is
supported only via execution of IA-32 binaries in compatibility model. We can also see that data types are
naturally aligned on Itanium™ - based systems. The details of the data alignment will be discussed in
the next few sections. On Linux for the Intel® Itanium™ Architecture, the little endian is native byte
order; however, it is possible to have big-endian processes in the system too.

Using defines and predefined macros properly

Part of the GNU guideline stresses that you should know how to use #define. Following are three
basic do’s and don’ts:

Do use #include to get system data types. If you need a system data type, include the system
header file.

Don’t declare data objects that you don’t define. In fact, ISO-C makes this a requirement.

Never declare a structure that the system defines.

These macros are defined by GNUPro and the Intel Linux compiler:

__unix, __linux, __ia64, __LP64, __ELF__, __GNUC__, __WORDSIZE (= 64)

It’s important to be certain you are testing the correct macros in trying to determine what’s true in a
particular environment. For example, if you’re trying to determine the size of a long or pointer, you
should not test __ia64 but rather __LP64. If you test __ia64, you could get an incorrect answer, since it’s
possible to support a 32-bit environment on the Intel® Itanium™ processor. In that case, testing __ia64
to determine the number of bits will generate the wrong answer. Testing LP64 will generate the correct answer.

If you’re testing for the Itanium™-based platform, you should be testing the instruction set architecture of the system, not determining whether it’s big or little endian or whether longs or pointers are 32 or 64 bits.

**Using ANSI const instead of #define Hex Constants**

Using ANSI const, the compiler can perform the necessary type checking, you will get warning if misuse is attempted. For example:

```c
#define OFFSET1 0xFFFFFFFF
#define OFFSET2 0x100000000
```

In 32-bit world, OFFSET1 is –1, OFFSET2 is 0 (most likely it causes an error); in 64-bit world, however, OFFSET1 is 4,294,967,295, OFFSET2 is 4,294,967,296.

Instead, you should ANSI’s type constant and properly qualify it with a `signed` or `unsigned`.

```c
const int OFFSET1 = 0xFFFFFFFF;
```

**Using Integer and Pointers Intelligently**

There are three more rules you should know about in dealing with integers, pointers, their sizes and alignment:

- Don’t cast pointers to long or int. Use `uint_ptr_t` to do the casting.
- Use `sizeof()` and `offsetof()` instead of hard-coded numbers to locate a piece of data from a structure.
- Use variables with type `size_t`, which can be found in all current header files, especially for C Runtime library calls. In ANSI/ISO C/C++, all ints and longs have been replaced with the appropriate version of `size_t`. When in doubt, look at the header file, and ask yourself “does this function return a `size_t`?” If so, declare a variable of `size_t` rather than type `int`.
- Use explicit sized types for external, on-the-wire, and shared memory structures.

For example: instead of

```c
struct on_disk { int reclen;
```

Use

```c
struct on_disk { int32_t reclen;
```

Here are more common examples:

- Variables receiving longs
Before porting:

long l;   // l is a 64-bit data type
int i;   // i is a 32-bit data type
long func(long l); // func returns a 64-bit value
i = (int)l + func(l);  // 64-bit to 32-bit assignment

After porting:

long l;   // l is a 64-bit data type
size_t i;   // i is a 64-bit data type
long func(long l); // func returns a 64-bit value
i = l + func(l);  // 64-bit to 64-bit assignment

- Variables receiving pointers

Before porting:

char p;   // &p is a 64-bit value
int i = &p   // 64-bit to 32-bit assignment

After porting:

char p;   // &p is a 64-bit value
uintptr_t i = &p  // 64-bit to 64-bit assignment

Packing, padding and alignment

Different data models will have different packing, padding and alignment rules. It’s important to gain a clear picture on how your structures are going to lay out in an Itanium™-based system’s memory and feel comfortable working with them.

If you are writing new software, or at least have this flexibility, sorting structures is a way to optimize for speed and also to ensure that you don’t have a conflict with the padding requirements of the Itanium™-based architecture. If you put the big and the highly aligned data object in the front; and put the less aligned objects toward the end, you will get better packing and higher efficiency in your structures. This reason is that the data can be found in the same cache line.

For example, the following structure is 12 bytes in 32-bit world and 24 bytes in the 64-bit world.

```c
Struct dim_t {
    int height;
    long width;
    int weight;
};
```
Reordering the fields, we can have

```c
struct dim_t {
__int3264 width;
int height;
int weight;
};
```

The new structure is 12 bytes in a 32-bit world as before, but it’s only 16 bytes in the 64-bit world. That’s 33% savings. Reordering the fields is a good weapon to fight the *code bloat*.

Also putting your pointers together, shorts together, and chars together will improve packing. If you are curious to know if the compiler had padded your structure to the alignment boundary, you can use gcc option –Wpadded.

Although we have mentioned above that data are naturally aligned, this can be overruled by #pragmas. For the application that shares the legacy structures, this feature comes in very handy.

**Truncation of long via double**

In the 32-bit world, double, with 52 significant bits can hold all bits in long. In the 64-bit world, precision of double is smaller than 64-bit integer. If the loss of significance is an issue to the application, use long double or REAL*10, if possible.

**Be aware of new, enhanced input/output format**

printf() and scanf() have been enhanced to support %p. This way you do not have to guess the size of a pointer or some nonportable code such as “%08IX”. Use %ld if you know you’re going to be printing a long data type. Another tenet of the GNU coding guideline: if you’ve got a long chain of #ifdef that’s trying to guess some property of the system, print out an error message instead of defaulting to the assumption that it’s 32 bits. For example:

Before porting:

```c
printf("%x", ptr_value);
//%x is 32-bit and will truncate the pointer
```

After porting:

```c
printf("%p", ptr_value);
//change to %p to display the full pointer value
```

**Using system calls parameters**

Use the API calls to get system parameters, and use predefined mnemonics in system include files. For example, use getpagesizek(), sysconf(_SC_SLK_TICK) and SEEK_END (not 2), INT_MIN (not -2147483648)
Inlining assembly code
Finally, if your application used any in-line assembly code, you may have to justify its existence. Compilers are becoming more and more sophisticated; any speed-based justification is becoming weaker and weaker. If you want to perform some processor-specific operations, you may want to find out if there is an equivalent standard runtime library to do the same thing. In short, inline assembly code is a porting hazard; you should try to get rid of it where possible.

Reading warnings sign
The Red Hat GNUPro compiler has been enhanced to check for various kinds of type capabilities. If you’ve got size incompatibilities or encounter alignment and padding warning messages or bad comparisons (sign with unsigned), you need to pay close attention as to whether or not the compiler is going to produce a surprising result.

Porting application from Windows to Linux for the Intel® Itanium™ Architecture
Understanding the difference in Programming Models
For the Itanium™-based platform, Windows* XP uses a data model called LLP64 or sometimes called P64. The main difference of this model from Linux’s LP64 is that the type long is 32-bit in Windows XP, and 64-bit under Linux. This is clearly a major hazard in porting application across these two platforms. However if you follow a few rules, you should avoid these problems.

Set goals for Linux functionality
When starting from Windows, it’s important to set realistic goals for Linux functionality. You should consider what features are most important and how you plan to support them on Linux. Further you should think about what functionality you should support with uncompromised and native performance, and which functionality you can sacrifice for limited budget and time-to-market constraints.

Having answered that question, you can consider if some of your less important, less resource consuming functionality can be quickly implemented by making use of the Windows portability library and even perhaps binary code execution environment. Products such as Wine*, Twine*, MainWin*, Wind/U* provide the Windows based applications with Microsoft compatible libraries in the Linux environment. While selectively using them for some non-core functionality will provide a cost and time-to-market advantage, you may also end up with a product offering that is not a true port of your application. Most likely the look-and-feel of your application will still be Windows-like, not Linux-like. Some points to keep in mind:
Use types with the same semantics in both Linux and Windows XP. short, int mean the same thing everywhere. signed and unsigned qualifiers are available everywhere. Use the newer data types common to both worlds. intptr_t, uintptr_t mean integer big enough to hold a pointer.

Find an appropriate GUI library toolkit in Linux

If you are suffering from negative repercussions from that “Windows look-and-feel on Linux”, you obviously need to find a good GUI library. Before you look for a graphics library package, find out if an off-the-shelf, GUI scripting tool will work for your application. TCL/TK* is such a candidate. It’s a mature, portable, and easy to use graphical scripting language. The current version 8.3.3 runs on Linux for the Intel® Itanium™ Architecture.

If GUI programming is warranted, there are quite a few choices when selecting a graphics library on Linux. GTK+ is an efficient, multi-platform widget set library for creating a graphical user interface object. It has the general look-and-feel of Motif but it contains more powerful, and highly sophisticated widgets such as file creation and color selection. GTK+ stands for GIMP Toolkits. GTK+ was originally written for GIMP, but gradually it evolved into generic GUI library, which has been used in many other applications, such as GNOME. For more information, you should visit http://www.gtk.org.

Qt (pronounced “cute”) is another multi-platform, C++ class library for GUI application. Like GTK+, it has the look-and-feel of Motif, but is much more platform independent and much easier to use than Motif. Unlike GTK, Qt uses an ingenious signal/slot mechanism for connecting user interaction with program functionality, providing an excellent framework for component-based programming and highly optimized graphical rendering. Qt has already established a solid position in the Linux community as the foundation of the KDE desktop interface effort, aimed at making Linux user-friendlier. For more information about Qt, you should read Matthias Kalle Dalheimer’s book, Programming with Qt.

Of course, if you want, you can also use Motif or its counterpart Lesstif. Both give you mature, multi-platform windows manager libraries that have existed for more than 10 years.

Achieving single source code by porting back to Windows platform

Once you have ported your Windows application to Linux, it does not mean that you have to maintain two different source trees. Many GUI libraries already have their Windows implementation. Red Hat also offers a library called cygwin.dll, which uses the C run time library and gives you a true and useful layer with Posix-like functions on top of Windows. By taking advantage of this, your newly ported Linux for the Intel® Itanium™ Architecture application may also run under Windows XP.
Step 4: Selecting development tools and toolkits for Linux

There is a wide range of choices for Linux software development tools for Itanium™-based systems. The first and foremost is the GNUPro toolkit from Red Hat. It can run on both IA-32 and Itanium™-based systems. Many people who have an Itanium™-based computer use this toolset.

<table>
<thead>
<tr>
<th>Tools Description</th>
<th>Native Tool Name</th>
<th>IA-32 cross Intel® Itanium™ Architectures Tool Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC Compiler</td>
<td>Gcc</td>
<td>ia64-linux-gcc</td>
</tr>
<tr>
<td>C++ Compiler</td>
<td>g++</td>
<td>ia64-linux-g++</td>
</tr>
<tr>
<td>GAS assembler</td>
<td>As</td>
<td>ia64-linux-as</td>
</tr>
<tr>
<td>GLD linker</td>
<td>Ld</td>
<td>ia64-linux-ld</td>
</tr>
<tr>
<td>Standalone simulator</td>
<td>Run</td>
<td>ia64-linux-run</td>
</tr>
<tr>
<td>Binary Utilities</td>
<td>ar</td>
<td>ia64-linux-ar</td>
</tr>
<tr>
<td></td>
<td>nm</td>
<td>ia64-linux-nm</td>
</tr>
<tr>
<td></td>
<td>objcopy</td>
<td>ia64-linux-objcopy</td>
</tr>
<tr>
<td></td>
<td>objdump</td>
<td>ia64-linux-objdump</td>
</tr>
<tr>
<td></td>
<td>ranlib</td>
<td>ia64-linux-ranlib</td>
</tr>
<tr>
<td></td>
<td>readelf</td>
<td>ia64-linux-readelf</td>
</tr>
<tr>
<td></td>
<td>size</td>
<td>ia64-linux-size</td>
</tr>
<tr>
<td></td>
<td>strings</td>
<td>ia64-linux-strings</td>
</tr>
<tr>
<td></td>
<td>strip</td>
<td>ia64-linux-strip</td>
</tr>
<tr>
<td>Gdb debugger</td>
<td>Gdb</td>
<td>ia64-linux-gdb</td>
</tr>
</tbody>
</table>

Intel also has a highly optimized Linux C/C++ software development toolset. Currently this toolset is available as a cross-development toolset on Linux for IA-32. In addition, Intel offers a Fortran compiler and a highly optimized math kernel library (MKL). This compiler is especially useful when you decide to use VTune™ to analyze the performance bottleneck of your application. Intel also offers an assembly instruction code dependency checking utility called DVLoc. For additional information on Linux development tools for Intel® Itanium™ processors, visit the Linux Microsite on the Intel Developer Services Web site.

In addition to the toolsets available from Red Hat and Intel, HP has Itanium™-based platform SDK toolset offerings. HP offers the Native User Environment (NUE), which is hosted on Linux for the IA-32
platform and emulate and execute code in an Itanium™ - based platform Linux environment. NUE is not just an Intel® Itanium™ processor simulator, but also provides the SDK tool chain, such as compiler, linker, assembler, C runtime libraries, and a runtime file system. With NUE from HP, the Itanium™ - based software developer can start a porting project on any IA-32 based Linux platform. You can download HP’s NUE at http://www.software.hp.com/products/LIA64/overview2a.htm

Step 5: Compiling the Application on Linux for the Intel® Itanium™ Architecture

Once you have rebuilt your application using a 32-bit Linux platform and cleaned up your code on Linux for the Intel® Itanium™ Architecture, rebuilding your application on the Itanium™ - based platform should become relatively easy. Assuming you are developing in a cross environment, the first step is to modify the makefile(s) for your build, as shown in the following example. If you are running 64-bit Linux on machines running Intel® Itanium™ processors, you may not need this step.

Example Code:

```plaintext
# IA-64 modified makefile for cross-compile
CROSSCOMPILE = IA-64-cygnus-linux-
TOOLPATH = /usr/IA-64-cygnus-linux/bin/
CC=$(TOOLPATH)$(CROSSCOMPILE)gcc
LD=$(TOOLPATH)$(CROSSCOMPILE)ld
CPP=$(CC) -E
AR=$(TOOLPATH)$(CROSSCOMPILE)ar
NM=$(TOOLPATH)$(CROSSCOMPILE)nm
STRIP=$(TOOLPATH)$(CROSSCOMPILE)strip
AS=$(TOOLPATH)$(CROSSCOMPILE)as
IA-64LIBS=-lc -ldb -lnss_files -lnss_dns -lresolv
IA-64CFLAGS= -O2
IA-64INC= -I /usr/IA-64-cygnus-linux/IA-64-cygnus-linux/include -static
IA-64LIBDIR = -L /usr/IA-64-cygnus-linux/IA-64-cygnus-linux/lib/
OBJDUMP=$(TOOLPATH)$(CROSSCOMPILE)objdump
GENKSYMS=/sbin/genksyms
INCLUDEROOT = /usr/IA-64-cygnus-linux/IA-64-cygnus-linux/include
INCLUDE = ${IA-64INC}
CFLAGS = ${IA-64CFLAGS} ${INCLUDE}
LDFLAGS = ${IA-64LDFLAGS}
LIBS = ${IA-64LIBS}
```
Now you are ready to compile the application using Linux for the Intel® Itanium™ Architecture.

First, clean the environment to remove all the object files and force a recompile to occur, and save the warnings in a text file:

```
make -f makefilename clean
make -f makefilename > warnings
```

The warnings text file will provide a base reference that is necessary because subsequent warnings will vary as you make changes to the source code.

Second, resolve the Itanium™-based platform warnings by making any necessary changes to the code. Note that these changes should not affect the functionality of the application on an Itanium™-based platform. Repeat this step until you have resolved all warnings.

Third, run regression tests on the IA-32 platform to ensure that the changes you made did not “break” anything. Modify your changes as necessary and repeat the regression tests until you are assured that your software runs cleanly.

That’s it. Now you have built an Itanium™-based application that you can recompile, optimize, and retest on Itanium™-based systems.

For more information . . .

This document has provided an introduction to porting Linux-based applications to the Itanium™-based platform. The goal is to provide you with a checklist for Itanium™-based application porting projects.

For more information on Linux for Itanium™-based systems, go to [http://www.linuxia64.org/](http://www.linuxia64.org/).

For more information on the Itanium™-based platform, go to:

For information on tools to port Linux to Itanium™-based systems, go to [http://developer.intel.com/design/Itanium/overview4a.htm](http://developer.intel.com/design/Itanium/overview4a.htm).

To learn more about the GNUPro* tool kit for IA-32 and Itanium™-based platforms, visit [http://www.redhat.com/software/tools/gnupro/](http://www.redhat.com/software/tools/gnupro/).

“GNU/Linux on IA-64” is an Open Source project. Visit [http://www.redhat.com](http://www.redhat.com) if you wish to contribute.