# Porting IRIX® Applications to SGI® Altix® Platforms: SGI ProPack™ for Linux®

007-4674-001

## CONTRIBUTORS

Written by Steven Levine

Illustrated by Chrystie Danzer

Production by Karen Jacobson

Engineering contributions by George Pirocanac

## COPYRIGHT



## LIMITED RIGHTS LEGEND



## TRADEMARKS AND ATTRIBUTIONS

# Record of Revision

| Version | Description |
|---|---|
| 001 | April 2004<br>Original publication |

# Contents

# Figures

# Tables

# Examples

# About This Guide

This publication provides information about porting an application to the SGI Altix platform

## Related Publications

The following SGI publications contain additional information that may be helpful for user's of SGI ProPack for Linux:

- *Linux Application Tuning Guide*
- *Linux Configuration and Operations Guide*
- *Linux Device Driver Programmer's Guide - Porting to SGI Altix Systems*
- *Linux Resource Administration Guide*
- *Message Passing Toolkit (MPT) User's Guide*
- *Performance Co-Pilot for IA-64 Linux User's and Administrator's Guide*
- *SGI Altix 3000 User's Guide*
- *SCSL User's Guide*
- *SGI ProPack for Linux Start Here*
- *XFS for Linux Administration*

## Obtaining Publications

You can obtain SGI documentation in the following ways:

- See the SGI Technical Publications Library at http://docs.sgi.com. Various formats are available. This library contains the most recent and most comprehensive set of online books, release notes, man pages, and other information.

- If it is installed on your SGI system, you can use InfoSearch, an online tool that provides a more limited set of online books, release notes, and man pages. With an IRIX system, select **Help** from the Toolchest, and then select **InfoSearch**. Or you can type `infosearch` on a command line.
- You can also view man pages by typing `man` <*title*> on a command line.

## Conventions

The following conventions are used throughout this publication:

| Convention | Meaning |
|---|---|
| `command` | This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures. |
| *variable* | Italic typeface denotes variable entries and words or concepts being defined. |
| **`user input`** | This bold, fixed-space font denotes literal items that the user enters in interactive sessions. (Output is shown in nonbold, fixed-space font.) |
| [] | Brackets enclose optional portions of a command or directive line. |
| ... | Ellipses indicate that a preceding element can be repeated. |
| `manpage`(*x*) | Man page section identifiers appear in parentheses after man page names. |

## Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, contact SGI. Be sure to include the title and document number of the manual with your comments. (Online, the document number is located in the front matter of the manual. In printed manuals, the document number is located at the bottom of each page.)

You can contact SGI in any of the following ways:

- Send e-mail to the following address:

  techpubs@sgi.com

- Use the Feedback option on the Technical Publications Library Web page:

  `http://docs.sgi.com`

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.

- Send mail to the following address:

  Technical Publications
  SGI
  1500 Crittenden Lane
  Mountain View, California 94043-1351

SGI values your comments and will respond to them promptly.

# Porting Overview

This document outlines the steps necessary to port an application from an IRIX system to the SGI Altix platform. We define *platform* as the set of software interfaces resting on a set of particular hardware, as shown in Figure 1-1. The *hardware* platform consists of the microprocessor and various other system devices, including disk drives, network connections, and system interconnects. Usually an application gains access to these devices through the operating system and system libraries which are called through a programming language. Other higher level services are provided by the layer called *middleware*.

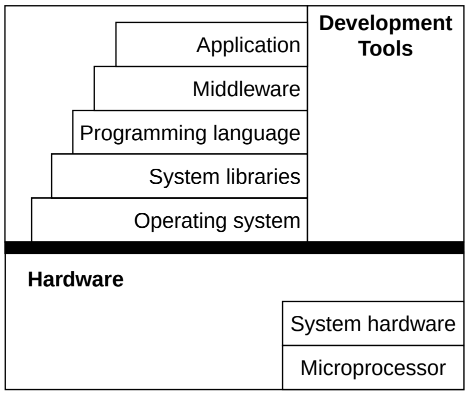


**Figure 1-1** Hardware and Software Platform

Table 1-1 provides an outline of two SGI platforms, the Origin3000 and the Altix. We can see that for the most part these systems have different sets of interfaces. We define porting as the act of adapting an application from one platform to another.

**Table 1-1** Platform Comparison

| System | Origin3000 | Altix |
|---|---|---|
| Processor | MIPS | IPF |
| System HW | NUMAlink | NUMAlink |
| OS | IRIX | ALE/ProPack (Linux) |
| System Libraries | libc, MPI, other | libc, MPI, other |
| Programming Languages | C, C++, Fortran, other | C, C++, Fortran, other |
| Middleware | various | various |

Given the large number of differences between the platforms, one might think that porting is essentially a rewrite of the application. Luckily, many of the differences between platforms are abstracted under standard programming languages and system libraries. At each level of the hierarchy there is a set of development tools that can aid in the process. For example, a C or Fortran program written for the Origin already has a lot of the porting issues solved through the use of standard compliance and features in the programming languages and library interfaces. A Java program (if written in100% pure Java) has even fewer porting issues to resolve.

Table 1-2 gives a comparison of various development tools and utilities available on Origin and Altix platforms.

**Table 1-2** Development Tools Comparison

| System | Origin3000 | Altix (Prop) | Altix (Open Source) |
|---|---|---|---|
| C/C++ Compiler | MIPSpro (cc) | ecc/icc | gcc |
| Fortran77 Compiler | (f77) | efc/ifort | g77 |
| Fortran90/95 Compiler | (f90) | efc/ifort | N/A |
| Text Debugger | (dbx) | idb | gdb |
| Kernel Debugger | kdb | kdb | kdb |

**Table 1-2** Development Tools Comparison **(continued)**

| System | Origin3000 | Altix (Prop) | Altix (Open Source) |
|---|---|---|---|
| GUI Debugger | ProDev Workshop | TotalView, various | ddd |
| App Perf Tools | SpeedShop | VTune | gprof |
| System Perf Tools | PCP | VTune/SGI Histx | PCP |
| Java Virtual Machine | 1.4.1 | 1.4.2 (BEA, Sun) | gjv |
| Array Services | 3.6 | 3.6 | |

Table 1-3 gives a comparison of various development libraries available on Origin and Altix platforms.

**Table 1-3** Development Libraries Comparison

| System | Origin 3000 | Altix (Prop) | Altix (Open Source) |
|---|---|---|---|
| Scientific Libraries | SCSL | SCSL, MKL, Goto BLAS | |
| MPI | MPT | MPT | MPICH, LAM |
| FFIO | Full support | Subset of IRIX support | |
| threads | sproc, pthreads | pthreads | pthreads |

At each level of the platform hierarchy there are possible porting issues. Table 1-4 summarizes the main ones at each level.

**Table 1-4** Platform Layer Porting Issues

| Platform Layer | Porting Issues |
|---|---|
| Processor | Assembly language, endianness |
| Other HW | Device drivers |
| OS | Differences in system calls and APIs |
| System libraries | Differences in APIs |
| Languages | Differences in ABIs, standards, adherence |

**Table 1-4** Platform Layer Porting Issues **(continued)**

| Platform Layer | Porting Issues |
|---|---|
| Middleware | Existence of different packages, different APIs |
| Development tools | Differences in the features and user interfaces |

This manual attempts to outline each of the issues and explain what it is and how to deal with it. It is best viewed online as many of items have hyperlinks into other SGI manuals and Internet web sites. It is by no means an exhaustive set, but it does highlight the major areas in some detail.

This remainder of this document is organized as follows:

- Chapter 2 describes endian order, and summarizes the differences in the byte order of a word in big-endian and little-endian systems.
- Chapter 3 discusses the issues that arise when trying to port 32-bit applications on the IRIX platform to the IPF 64-bit ABI on the Altix platform.
- Chapter 4 describes the similarities and differences between the development tools environments found on IRIX and ProPack (Linux), concentrating on the compilation process and tools available for Altix.
- Chapter 5 outlines additional development tools which are mainly used after an application has been built (or to automate the build process).
- Chapter 6 discusses the differences in Message Passing Toolkit (MPT) support between IRIX and Linux systems.
- Chapter 7 outlines differences between the POSIX threads (Pthreads) implementations on IRIX 6.5 and the latest version of ProPack.
- Chapter 8 provides a list of issues that you may need to address when porting an application from an IRIX to a Linux system.
- Chapter 9 gathers some frequently asked questions regarding application porting and provides the answers.
- Appendix A notes the standard C libraries (`libc`) that are available on IRIX but missing on Linux.

# Endian Order

One of the major issues that may arise when porting an application is endian order. This chapter summarizes the differences in byte order between big endian systems and little endian systems.

## A Case of Endianness

A big endian machine (such as MIPS/IRIX) will lay out a word in memory such that the highest order byte will be at the lowest address. For example the 32-bit word 0x12345678 will be laid out on a big endian machine as follows:

| Memory offset | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| Memory content | 0x12 | 0x34 | 0x56 | 0x78 |

If we view 0x12345678 as two half words, 0x1234 and 0x5678, we would see the following in a big endian machine:

| Memory offset | 0 | 2 |
|---|---|---|
| Memory content | 0x1234 | 0x5678 |

On a little endian machine (IA64/Altix) the highest order byte is at the highest address. So our examples above would look like the following:

| Memory offset | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| Memory content | 0x78 | 0x56 | 0x34 | 0x12 |

Similarly, the two half-words 0x1234 and 0x5678 would look like the following:

| | | |
|---|---|---|
| Memory offset | 0 | 2 |
| Memory content | 0x3412 | 0x7856 |

It is important to realize that individual bytes are addressed in the same order on either machine. Thus, the following stream would appear in this same order on both a big endian and a little endian system:

| | | | |
|---|---|---|---|
| 0x12 | 0x34 | 0x56 | 0x78 |

## Examples

The following examples illustrate the difference in byte order between big endian and little endian machine.

The following C program will print out “big endian” when compiled and run on a big endian machine and “little endian” when compiled and run on a little endian machine.

**Example 2-1** C Program Illustrating Endian Order

```
#include <stdio.h>

main()
{
   int i = 0x12345678;

   if ( *(char *)&i == 0x12 )
      printf(“Big endian\n”);
   else if ( *(char *)&i == 0x78 )
       printf(“Little endian\n”);
}
```

The following Fortran program, when compiled, creates a file, `fort_7` that contains the string “ABCDEFGHIJ”:

**Example 2-2** Fortran Program Illustrating Endian Order

```
        program yy

        character*10 str
        integer strint(2)
!        equivalence (str,strint(1))

        str = "ABCDEFGHIJ"
!       print 10, str,strint(1),strint(2)
! 10    format(2x,a10,2x,z20,2x,z20)

        open(unit=7,form='unformatted',file="fort_7",status='new')
        write(unit=7) str
        close(unit=7)

        end
```

After compiling this program, note the results of a dump from the `od` command:

```
% yy
% od -hc fort_7
0000000 000a 0000 4241 4443 4645 4847 4a49 000a
         \n  \0  \0  \0   A   B   C   D   E   F   G   H   I   J  \n  \0
0000020 0000
         \0  \0
0000022
```

This program was compiled on a little endian machine. The individual bytes 'A' 'B' 'C' etc. are in order. Notice how when read as words the 'A' & 'B' are swapped to 4241. When we get to the end we see the delimiter as the bytes 0a 00 00 00 and the two swapped halfwords 000a and 0000 which corresponds to a word value of 0x0000000a or 10 (decimal).

On a big endian machine the delimiter word would be as follows in bytes:

00 00 00 0a

This is what we see when we run this on a big endian machine OR set the endian environment variable mode to be big endian on an Altix, as in the following example. Note the order of the byte characters \0 \0 \0 \n.

```
% setenv  F_UFMTENDIAN big

% rm fort_7
% yy
% od -hc fort_7
0000000 0000 0a00 4241 4443 4645 4847 4a49 0000
```

```
         \0  \0  \0  \n   A   B   C   D   E   F   G   H   I   J  \0  \0
0000020 0a00
         \0  \n
0000022
```

Because we are on a little endian box, the words are swapped. So in this case 00 00 and 00 0a become 0000 and 0a00

# 64-bit ABI Porting Issues

In this section we examine the issues that arise when trying to port 32-bit application from an IRIX to an Altix Platform.

---

**Note:** If your source code has already been ported to the 64-bit ABI on IRIX you can skip this section although you should check to see if you are using `#ifdef` operations to conditionally control compilation for 32-bits or 64-bits. These `#ifdef` operations may need to be modified to use symbols from the header files found on Altix rather than those under IRIX.

---

## 32-bit and 64-bit Differences

Unlike MIPS/IRIX which has both n32 and o32, Altix has no 32-bit ABIs; under Altix, all applications follow the 64-bit ABI, IPF. Like the 64-bit ABI on MIPS, the IPF ABI is termed LP64; the C long integer (L) and pointer (P) data types are 64-bits. On a 32-bit ABI both of those types are 32-bits. Table 3-1 summarizes the C data type sizes for both the 32-bit and 64-bit ABIs.

**Table 3-1** C data type sizes in 32-bit and 64-bit ABI

| Type | Size in 32-bit ABI | Size in 64-bit ABI |
|---|---|---|
| char | 8 bits | 8 bits |
| short | 16 bits | 16 bits |
| int | 32 bits | 32 bits |
| long | 32 bits | 64 bits |
| long long | 64 bits | 64 bits |
| void * | 32 bits | 64 bits |

**Table 3-1** C data type sizes in 32-bit and 64-bit ABI **(continued)**

| Type | Size in 32-bit ABI | Size in 64-bit ABI |
|---|---|---|
| float | 32 bits | 32 bits |
| double | 64 bits | 64 bits |
| long double | 128 bits | 128 bits |

It turns out that the issues one encounters when porting from 32-bits to 64-bits are based on faulty assumptions about integers, long integers and pointers all being the same size (32-bits). These assumptions can be explicit, such as use of 32-bit variables to hold 64-bit types, or they could be more subtle, such assumptions about the way certain structures are laid out and aligned. The following is a list of such faulty assumptions:

sizeof(long) == sizeof(int)

Code that is written specifically for 32-bits often interchanges long integers and regular integers without consequences. Under LP64, however, such code could introduce truncation or improper sign extension.

sizeof(void *) == 4

This assumption is analogous to the previous one. But mappings to external data structures should seldom be a problem, since the external definition should also assume 64-bit pointers in the LP64 model.

faulty constants (i.e. -1 = 0xffffffff)

The change in type sizes may yield some surprises related to constants. You should be particularly careful about using constants with the high-order (sign) bit set. For instance, the hex constant 0xffffffff yields different results in the expression:

```
long x;
... ( (long) ( x + 0xffffffff ) ) ...
```

In both models, the constant is interpreted as a 32-bit unsigned int, with value 4,294,967,295. In the 32-bit model, the addition result is a 32-bit unsigned long, which is cast to type long and has value x-1 because of the truncation to 32 bits. In the LP64 model, the addition result is a 64-bit long with value x+4,294,967,295, and the cast is redundant.

arithmetic assumptions

Related to some of the above cases, code which does arithmetic (including shifting) which may overflow 32 bits, and assumes particular treatment of the overflow (for example, truncation), may exhibit different behavior in the LP64 model, depending on the mix of types involved (including signedness).

Similarly, implicit casting in expressions which mix int and long values may behave unexpectedly due to sign/zero extension. In particular, remember that integer constants are sign or zero extended when they occur in expressions with long values.

Once identified, each of these problems is easy to solve. Change the relevant declaration to one which has the desired characteristics in both target environments, add explicit type casts to force the correct conversions, use function prototypes, or use type suffixes (for example, \`l' or \`u') on constants to force the correct type.

printf() format assumptions

Code that has been tailored to the 32-bit ABI has diagnostics that rely on printf using the %x formatting type to print out pointer values. Under LP64, this formatting would only print 32-bits of the pointer value. To be truly portable the %p format should be used.

## Writing C Code Portable to 64-Bit Platforms

The key to writing new code which is compatible with the 32-bit and LP64 data models described is to avoid those problems described above. Since all of the assumptions described sometimes represent legitimate attributes of data objects, this requires some tailoring of declarations to the target machines' data models.

We suggest observing the guidelines in the following procedure to produce code without the more common portability problems. They can be followed from the beginning in developing new code, or adopted incrementally as portability problems are identified.

1. Use a header file that can be included in each of the program's source files, and defines a type (with a `typedef` statement) for each specific integer data size required. That is, where exactly the same number of bits is required on each target, define a signed and unsigned type, as in the following example.

   ```
   typedef signed char int8_t
   typedef unsigned char uint8_t
   ...
   typedef unsigned long long uint64_t
   ```

   On an Altix system this header file is `/usr/include/stdint.h.`

2. If you require a large scaling integer type, that is, one which is as large as possible while remaining efficiently supported by the target, define another pair of types, for example:

   ```
   typedef signed long intscaled_t
   typedef unsigned long uintscaled_t
   ```

   If you require integer types of at least a particular size, but chosen for maximally efficient implementation on the target, define another set of types, similar to the first but defined as larger standard types where appropriate for efficiency.

   Having included the above header file, use the new typedef'ed types instead of the standard C type names. You need (potentially) a distinct copy of this header file (or conditional code) for each target platform supported. As a special case of this, if you are providing libraries or interfaces to be used by others, be particularly careful to use these types (or similar application specific types) chosen to match the specific requirements of he interface. Also in such cases, you should choose the actual names used to avoid name space conflicts with other libraries doing the same thing. If this is done carefully, your clients should be able to use a single set of header files on all targets.

3. Be careful that constants are specified with appropriate type specifiers so that they extend to the size required by the context with the values that you require. Bit masks can be particularly troublesome in this regard:avoid using constants for negative values. For example, 0xffffffff may be equivalent to a -1 on 32-bit systems, but it is interpreted as 4,294,967,295 (signed or unsigned) on 64-bit systems. The `/usr/include/stdint.h` header file provides definitions to facilitate this conversion.

4. Defining constants that are sensitive to type sizes in a central header file may help in modifying them when a new port is done. Where `printf`()/`scanf`() are used for objects whose types were defined with different `typedef` statements among the targets you must support, you may need to define constant format strings for each of the types defined in step 1,

   For example, you may need to define the following constant format strings:

   ```
   #define _fmt32 “%d”
   #define _fmt32u “%u”
   #define _fmt64 “%ld”
   #define _fmt64u “%lu”
   ```

   On Altix platforms the `/usr/include/inttypes.h` header file provides `printf`()/`scanf`() format extensions to standardize these practices.

## Writing Fortran Code Portable to 64-Bit Platforms

This section describes which sections of your Fortran source code you need to modify to port to a 64-bit system.

Standard Fortran code should have no problems, but the following areas need attention:

- Code that uses `REAL*16` could get different runtime results due to additional accuracy in the `QUAD` libraries on IRIX. (There are no equivalent libraries on Altix.)
- Code compiled at high optimization levels by the MIPSpro and Intel IPF compilers may yield different answers due to operations being ordered (and reordered) differently by the compilers. The compilers may also perform constant folding differently.
- Integer variables which were used to hold addresses in 32-bit applications need to be changed to `INTEGER*8`.
- C interface issues may need to be addressed (Fortran passes by reference so addresses need to be 64-bits).
- The `%LOC` extension returns 64-bit addresses under the 64-bit ABI.
- The `%VAL` extension passes 64-bit values under the 64-bit ABI.

## Examples of Fortran Portability Issues

The following examples illustrate the variable size issues outlined above:

**Example 3-1** Changing Integer Variables

Integer variables used to hold addresses must be changed to INTEGER*8.

32-bit code:

```
integer iptr, asize
iptr = malloc(asize)
```

64-bit code:

```
integer*8 iptr, asize
iptr = malloc(asize)
```

**Example 3-2** Enlarging Tables

Tables which hold integers used as pointers must be enlarged by a factor of two.

32-bit code:

```
integer tableptr, asize, numptrs
numptrs = 100
asize = 100 * 4

tableptr = malloc(asize)
```

64-bit code:

```
integer numptrs
integer*8 tableptr, asize

numptrs = 100
asize = 100 * 8

tableptr = malloc(asize)
```

**Example 3-3** Storing `%LOC` Return Values

`%LOC` returns 64-bit addresses. You need to use an `INTEGER*8` variable to store the return value of a `%LOC` call.

```
      INTEGER*8 HADDRESS
C determine memory location of dummy heap array
      HADDRESS = %LOC(HEAP)
```

**Example 3-4** Modifying C Routines Called by Fortran

C routines which are called by Fortran where variables are passed by reference must be modified to hold 64-bit addresses. Typically, these routines used ints to contain the addresses in the past. For 64-bit use, at the very least, they should use long ints. There are no problems if the original C routines simply define the parameters as pointers.

Fortran:

```
call foo(i,j)
```

C:

```
foo_( int *i, int *j) or at least
foo_( long i, long j)
```

**Example 3-5** Declaring Fortran Arguments as long ints

Fortran arguments passed by %VAL calls to C routines should be declared as long ints in the C routines.

Fortran:

```
call foo(%VAL(i))
```

C:

```
foo_( long i )
```

**Example 3-6** Changing Argument Declarations in Fortran Subprograms

Fortran subprograms called by C where long int arguments are passed by address need to change their argument declarations.

C:

```
long l1, l2;
foo_(&l1, &l2);
```

Fortran:

```
subroutine foo(i, j)
integer*8 i,j
```

# Compiler and Development Tools

This chapter and Chapter 5 describe the similarities and differences between the development tools environments found on IRIX and ProPack (Linux) on Altix systems. This chapter concentrates on the compilation process and tools available for Altix systems while Chapter 5 outlines other development tools. Both chapters provide links to more detailed information available on the Web as well as in other SGI Technical Publications.

## Development Tool Chain

The development process can be thought of as a chain of processes aided by a variety of software tools. shows this in table form.

**Table 4-1** Development Process

| Activity | Tools | IRIX versions | Linux versions |
| --- | --- | --- | --- |
| Source code development | Editors | `vi`, `emacs`, `jot`, etc | `vi`, `emacs`, etc. |
| Executable creation | Compilers | `cc`, `CC`, `f77`, `f90` | `ecc`, `gcc`, `efc/ifort`, `g77` |
| Object file creation | Assemblers | `as` | `ias`, `as` |
| Linkage | Linker | `ld` | `ld` |
| Archiving | Archiver | `ar` | `ar` |
| Object file inspection | Object tools | `elfdump`, `dwarfdump` | `objdump` |
| Debugging | Debuggers | `dbx`, `cvd` | `gdb`, `idb`, `ddd`, `DDT` |
| Performance analysis | Profilers | `SpeedShop`, `perfex` | `VTUNE`, `perfmon`, `histx` |
| Automation | Make | `make`, `smake`, `pmake` | `gmake` |
| Environment configuration | Scripts/tools | modules | modules |

# Editors

A variety of editors are supported on both IRIX and Linux platforms. The two most common UNIX editors, `vi`(1) and `emacs`(1) are available on both platforms.

# Compilers

Compilers for Altix fall into two major categories:

- Proprietary compilers from Intel
- Open Source compilers from the Free Software Foundation (GNU)

## Intel Compilers

Intel provides compilers that support C/ C++, and Fortran95. For information, see http://www.intel.com/software/products/compilers/linux/

As of this writing Intel has delivered the 8.0 Compilers which offer a new Fortran95 front-end compatible with Compaq Visual Fortran 6.6.

Table 4-2 shows the various releases of the Intel compilers. It should be noted that the 8.0 Fortran compiler is not binary compatible with the earlier 7.*x* compilers and you will need to recompile your application to work with libraries compiled with Intel Fortran95 version 8.0.

**Table 4-2** Intel Compiler Versions

| Version | Date of Release |
|---|---|
| 7.0 Compilers | November 2002 |
| 7.1 Compilers | March 2003 |
| 8.0 Compilers | December 2003 |

The compilers themselves can be installed in a modules environment (also available on IRIX), which allows several versions to co-exist.

The following script called `run_latest` shows an example of the use of modules. This script sets up a C-shell modules environment where the 8.0 compilers are configured to be the default (the 8.0 Intel compilers were installed in a module called `intel-compilers-8`):

**Example 4-1** Script to Set Up C-shell Modules Environment

```
%cat run_latest
source /sw/com/modules/init/csh       # to set up module command
module avail                          # to display what are on a system
module load intel-compilers-8         # make 8.0 default compilers
```

The example `/sw/com/modules/init/csh` script initializes the modules environment with locations where the various modules are installed.

**Example 4-2** Script to Initialize Modules Environment

```
% cat /sw/com/modules/init/csh
# Generated automatically from csh.in by configure.
if ( $?tcsh ) then
        set modules_shell = “tcsh”
else
        set modules_shell = “csh”
endif

set exec_prefix=”/sw/com/modules”

if ( $?histchars ) then
  set histchar = `echo $histchars | cut -c1`
  set _histchars = $histchars
  alias module ‘unset histchars; \
eval `$exec_prefix/bin/modulecmd $modules_shell ‘$histchar’*`; \
set histchars = $_histchars’
  unset histchar
else
  alias module ‘eval `$exec_prefix/bin/modulecmd $modules_shell \!*`’
endif

setenv MODULESHOME /sw/com/modules

if (! $?MODULEPATH ) then
  setenv MODULEPATH /sw/com/modulefiles
endif

if (! $?LOADEDMODULES ) then
  setenv LOADEDMODULES
endif
```

The command names of the compiler are given below. They are different from the `cc`, `CC`, `f77` and `f90` compiler commands available on IRIX.

The `ecc` command can be used on both c and C++ filenames, while the `efc` command works on a variety of Fortran suffixes. Example 4-3 summarizes the command line syntax for the Intel 7.x Compilers.

**Example 4-3** Compiler Command Line Syntax For Intel Version 7.x Compilers

```
ecc [ option(s) ] filename.{c|C|cc|cpp|cxx|i}

efc [ option(s) ] filename.{f|for|ftn|f90|fpp}
```

For the Intel 8.0 compilers, the command names have been changed to `ifort` and `icc` respectively though the old names will also be accepted. (A warning will be generated however, and Intel reserves the right to stop supporting the 7.x command names in a future release.) Example 4-4 summarizes these commands.

**Example 4-4** Compiler Command Line Syntax For Intel Version 8.0 Compilers

```
icc [ option(s) ] filename.{c|C|cc|cpp|cxx|i}

ifort [ option(s) ] filename.{f|for|ftn|f90|fpp}
```

## GNU Compilers

The GNU compilers are provided by the Free Software Foundation. They have been ported to a variety of architectures and offer easy migration to and from other platforms. The Linux kernel itself and various other system utilities on ProPack are compiled with `gcc`.

The URL http://www.gnu.org/directory/gcc.html is the top level web site for the `gcc` compilers.

The GNU compiler command names are different from the MIPSpro compilers on IRIX and the Intel compilers, though they are standard on all GNU supported platforms. Example 4-5 summarizes these commands.

**Example 4-5** Compiler Command Line Syntax for the GNU Compilers

```
gcc [ option(s) ] filename.{c|C|cc|cxx|m|i}

g++ [ option(s) ] filename.{c|C|cc|cxx|m|i}

g77 [ option(s) ] filename.{f|for|fpp|F}
```

# Standards Support

Compilers provide common programming environments through the support of standards. Non-standard features are called extensions. Normally vendors provide features outside of the standards to provide capabilities that are not possible to achieve with standard compliant code. Another reason for extensions are to provide access to unique performance enhancing capabilities. Finally extensions are often provided to ensure capabilities with obsoleted features that have been removed from more current standards.

## C Language Standard Support

The Intel compilers support the new ANSI C Standard (1999) or C99 with the `-c99` flag (or by setting `-std=c99`). This is on by default on Altix whereas under MIPSpro 7.4.x on IRIX the `-c99` flag or `c99` command had to be used. Also supported is the older ANSI Standard (1989) `c89` as well as Amendment 1 (1990).

The GNU Compilers offer compatibility with the C89 standard and limited support of C99. For information, see http://gcc.gnu.org/onlinedocs/gcc-3.3.2/gcc/Standards.html#Standards and http://gcc.gnu.org/gcc-3.3/c99status.html (which describes what features of c99 are supported when `-std=c99` is used.

The GNU Compilers also offer a variety of extensions to the C language These features are often used in open source software and can be thought of as a standard in itself. For more information see: http://gcc.gnu.org/onlinedocs/gcc-3.3.2/gcc/C-Extensions.html#C%20Extensions.

The Intel C compiler provides a large level of compatibility with `gcc`. In general one can freely mix object files compiled with the Intel compilers and those built with `gcc`. For more information see: Intel Compilers for Linux: Compatibility with GNU Compilers at http://www.intel.com/software/products/compilers/techtopics/LinuxCompilersCompatibility702.htm.

Table 4-3 summarizes the standards compliance by the different compilers.

**Table 4-3** C Language Standard Support Summary

| Standard | Intel | MIPSpro | GNU |
|---|---|---|---|
| C89 | X | X | X |
| C99 | X | X | partial |
| GNU extensions | many | some | X |
| OpenMP1.0 | X | X | |
| OpenMP2.0 | X | X | |

## C++ Language Standard Support

The Intel compilers support the ANSI C++ Standard (1998) with the exception of the export keyword. In general this is a superset of the ANSI standard compliance provided by the MIPSpro 7.4.x C++ compiler (under the default `-LANG:std=on` setting).

The GNU compilers offer a variety of extensions to standard C++. For more information see:
http://gcc.gnu.org/onlinedocs/gcc-3.3.2/gcc/C-Extensions.html#C++%20Extensions

## Fortran Language Standard Support

The Intel compilers support the ANSI FORTRAN77, Fortran90 and Fortran 95 standards. They also provide support for a new Fortran2003 feature. In addition, the Intel compilers provide the ability to handle both big and little endian files. This allows the program to handle data files generated on IRIX (big endian) system. A set of environment variables control whether files are read in big or little endian mode, as shown in the following examples:

```
% setenv F_UFMTENDIAN big # READS and WRITES big endian files
```

```
% setenv F_UFMTENDIAN big:10 # perform conversion only on unit 10
```

On IRIX there were two different Fortran compiler products. MIPSpro FORTRAN77 provided support for the FORTRAN77 standard as well as various VAX extensions. MIPSpro Fortran90, despite its name, provided support for almost all of Fortran95 in addition to Fortran90 (as it name implies) and FORTRAN77. The IRIX man page `f77.f90.difs`(5) provided detailed information about the differences in language support between MIPSpro FORTRAN77 and MIPSpro Fortran90.

The GNU compilers provide FORTRAN77 support, some Fortran90 features and some extensions. For more information see:
http://gcc.gnu.org/onlinedocs/gcc-3.3.2/g77/Language.html#Language

Table 4-4 summarizes the standards compliance and provision of extensions by the various compilers

**Table 4-4** Fortran Language Standard Support Summary

| | **MIPSpro FORTRAN77** | **MIPSpro F90** | **Intel Fortran95** | **g77** |
|---|---|---|---|---|
| Fortran2003 | | | [a] | |
| Fortran95 | | X[b] | X | |
| Fortran90 | | X | X | [a] |
| Vax Extensions | X | | X | |
| `%loc` | X | | X | X |
| `%val` | X | | X | X |
| `%fill` | | | X | |
| `%ref` | X | | X | |
| `TIME` intrinsic | X | | X | X |
| `ACCESS='KEYED'` | X | | | |
| CRAY pointers | X | X | X | |
| `VOLATILE` keyword | | X | X | |
| FORTRAN77 | X | X | X | X |
| Hollerith constants | X | X | X | |
| Cross Endian support | | | X | |
| POSIX Interfaces | [a] | X | X | |
| Options for default size of integer and real (`-i8`, `-r8`) | X | X | X | |

a. Some functionality is provided

b. Some caveats apply

### OpenMP Standard Support

OpenMP is a standard set of programming directives, application program interfaces and environment variables that provide a portable interface for developing parallel applications on shared memory systems. For more information see http://www.openmp.org/.

Both the MIPSpro 7.4 Fortran90 and Intel Fortran compilers support the OpenMP 2.0 standard with some restrictions. The MIPSpro compilers will serialize nested parallel regions. The Intel compiler does not support the WORKSHARE directive. Also, the MIPSpro compilers supported proprietary data distribution directives which are not supported by the Intel compilers.

In addition to the standard OpenMP directives and environment variables, the Intel Compilers support the following environment variables as extensions:

`KMP_LIBRARY`
: Selects the OpenMP run-time library throughput

`KMP_STACKSIZE`
: Sets the number of bytes to allocate for each parallel thread. (Default is 4megabytes) This is similar to the `MP_SLAVE_STACKSIZE` environment variable under MIPSpro.

The GNU compilers do not support any OpenMP directives or environment variables.

## Compiler Options

Although there are many differences between the set of flags that each compiler supports, there are also flags common to all three compilers (MIPSpro, Intel, GNU). Table 4-5 summarizes this list.

**Table 4-5** Common Compiler Flags on IRIX and Linux (Both Intel and GNU)

| | |
|---|---|
| `-ansi` | Support all ANSI standard C programs |
| `-c` | Compiles but does not link. Creates a `.o` file. |
| `-D`*macro* | Define macro on command line |
| `-I` *dir_name* | Searches for include files in *dir_name* |

**Table 4-5** Common Compiler Flags on IRIX and Linux (Both Intel and GNU) **(continued)**

| | |
|---|---|
| `-g` | Produces symbolic information for debugging |
| `-help` | Print list of compiler options. (`--help` with gcc) |
| `-L` *dir_name* | instruct linker to search *dir_name* for libraries. |
| `-M` | Generate makefile dependency lines for each source file. |
| `-o` *file_name* | Creates output file with *file_name*.. |
| `-O` | Invokes default optimization level. |
| `-S` | Creates assembly language (`.s`) file. |
| `-U`*macro* | Undefine macro. |
| `-v` | Verbose. Prints the passes as they execute with their arguments and their input and output files. |
| `-W` | Suppress warning information. |

The MIPSpro and Intel compilers also provide many of the same types of functionality, often through the use of similar but slightly different flags. Table 4-6 summarizes these compiler flags.

**Table 4-6** Similar Compiler Flags on MIPSpro and Intel compilers

| | **MIPSpro** | **Intel** |
|---|---|---|
| Automatic parallelization | `-apo` | `-parallel` |
| Check array bounds | `-C` | `-CB` |
| Turn warnings into errors | `-diag_error` | `-Werror` |
| Use Fortran preprocessor | `-ftpp` | `-fpp` |
| Interprocedural optimization | `-ipa` | `-ipo` |
| Interpret OpenMP directives | `-mp` | `-openmp` |
| Select optimizations that enhance performance | `-Ofast` | `-fast` |
| Set maximum number of times to unroll loops | `-OPT:unroll_times_max=`*n* | `-unroll`*n* |
| Provide compiler version | `-V` (with no file) | `-V` |

There is a small set of flags which are common to the MIPSpro and Intel compilers but which have completely different meanings. These flags are described in Table 4-7.

**Table 4-7** Conflicting Compiler Options

| Flag | MIPSpro meaning | Intel meaning |
|---|---|---|
| `-C` | Check array bounds. | Preserve comments in preprocessed source output. |
| `-mp` | Cause the compiler to recognize multiprocessor directives. | Restrict optimizations in floating point applications to ensure arithmetic conforms to IEEE standards. |
| `-static` | Statically allocate all local variables. | Use the static library for linking |

Finally, there are those flags that are available only on the Intel compilers. Table 4-8 provides a partial list of these flags.

**Table 4-8** Intel-only Flags

| | |
|---|---|
| `-auto` | Direct all local variables to be automatic (Fortran) |
| `-EP` | Direct the preprocessor to expand source and output it to standard output, but `#line` directives are not included in the output. (`-EP` is equivalent to `-E -P`.) |
| `-convert` *keyword* | Specifies format of unformatted files containing numerical data. |
| `-fno-alias` | Use pointers with no aliasing in C. |
| `-ftz` | Force flushing of denormalized results to zero. |
| `-IPF_Fltacc` | Disable optimizations that affect floating-point accuracy. |
| `-opt_report`<br>`-opt_report_file` *file* | Generate an optimization report and direct it to `stderr` (or to *file* if `-opt_report_file` is specified. |
| `-safe_cray_pointer` | No aliasing for Cray pointers. (Fortran) |
| `-stack_temps` | Allocate arrays on stack. (Fortran) |
| `-Wp64` | Print diagnostics for 64-bit porting. |

## Compiler Directives

Table 4-9 provides s a subset of directives that are supported by the Intel compilers which may aid in tuning or debugging applications:

The Fortran form for these directives is the following:

`cdir$` *directive_name*

The C form for these directives is the following:

`#pragma` *pragma_name*

The *directive_name* and *pragma_name* variables are the same. For brevity, Table 4-9 provides the Fortran form only.

**Table 4-9** Compiler Directives for Tuning and Debugging

| | |
|---|---|
| `cdir$ ivep` | Ignore vector dependencies |
| `cdir$ swp` | Try to software pipeline an inner loop |
| `cdir$ noswp` | disable software pipelining |
| `cdir$ loop count` *N* | Software pipelining hint |
| `cdir$ distribute point` | Split large loop |
| `cdir$ unroll` *N* | Unroll loop *N* times |
| `cdir$ nounroll` | Do not unroll loop |
| `cdir$ prefetch` *A* | Prefetch Array *A* |
| `cdir$ noprefetch` *A* | Do not prefetch array *A* |

For more information about these and other directives supported by the Intel Compilers see “Chapter 14 Directive Enhanced Compilation, Intel Fortran Language Reference” at http://www.intel.com/software/products/compilers/flin/docs/for_lang.htm.

## Assemblers

Both Intel and GNU provide assemblers for IA64. The Intel assembler is called `ias;` the GNU assembler is called `as`. The Intel compiler normally bypasses calling the assembler by directly compiling into a an object file, while `gcc` normally creates an assembly language file (`.s`) and then calls the assembler to assemble it into an object file.

Since assembly language programming is inherently unportable, you will need to entirely rewrite assembly language code when porting from IRIX to Altix. A clear understanding of the machine architecture is required.

More information about the Intel assembler can be found at: http://www.intel.com/software/products/opensource/tools1/tol_white.htm.

More information about the GNU assembler can be found by consulting its man page:

```
%man as
```

## Linker

The GNU linker (`/usr/bin/ld`) is used by both the Intel compiler and `gcc` to combine a number of object files and archives into an executable. It supports the standard `-L`*directory* and `-l`*name* options to specify which directory to search for `libname.a` or `libname.so`, as well as the `-o`*executable* option to name the resulting executable file.

The `ld` linker also supports a variety of options which in general, are different from the MIPSpro linker. (It should also be mentioned that the GNU linker is more sensitive to the order of libraries given on the command line than its MIPSpro counterpart.)

For options whose names are a single letter, option arguments must either follow the option letter without whitespace, or be given as separate arguments immediately following the option that requires them.

For options whose names are multiple letters, either one dash or two can precede the option name; for example, `-trace-symbol` and `--trace-symbol` are equivalent.

If the linker is being invoked by either the Intel compiler or `gcc` then all of the linker command line options should be provided and prefixed by `-Wl`.

For more information on the GNU linker, consult the `ld`(1) man page.

# Additional Development Tools

As described in Chapter 4, the development process can be thought of as a chain of processes aided by a variety of software tools. Table 5-1 shows the development tool chain in table form.

Chapter 4 described the compilation process and tools available for Altix systems. This chapter outlines other development tools which are mainly used after an application has been built (or to automate the build process).

**Table 5-1** Development Process

| Activity | Tools | IRIX versions | Linux versions |
|---|---|---|---|
| Source code development | Editors | `vi`, `emacs`, `jot`, etc | `vi`, `emacs`, etc. |
| Executable creation | Compilers | `cc`, `CC`, `f77`, `f90` | `ecc`, `gcc`, efc/`ifort`, `g77` |
| Object file creation | Assemblers | `as` | `ias`, `as` |
| Linkage | Linker | `ld` | `ld` |
| Archiving | Archiver | `ar` | `ar` |
| Object file inspection | Object tools | `elfdump`, `dwarfdump` | `objdump` |
| Debugging | Debuggers | `dbx`, `cvd` | `gdb`, `idb`, `ddd`, `DDT` |
| Performance analysis | Profilers | `SpeedShop`, `perfex` | `VTUNE`, `perfmon, histx` |
| Automation | Make | `make`, `smake`, `pmake` | `gmake` |
| Environment configuration | Scripts | modules | modules |

## Archiver and Other Object file Tools

The archiver (`ar`) maintains groups of files as a single archive file. Generally, you use this utility to create and update library files that the linker uses, however, you can use the archiver for any similar purpose. On Linux, `ar` is the GNU archiver. The archiver flags are similar on IRIX and Linux; Table 5-2 summarizes their common flags.

**Table 5-2** IRIX and Linux Common Archiver Options

| | |
|---|---|
| `-d` | Deletes specified object |
| `-m` | Moves specified object to the end of the archive |
| `-p` | Prints the specified members of the archive to `stdout` |
| `-q` | Appends specified object to the end of the archive |
| `-r` | Replaces an earlier version of the object in the archive |
| `-t` | Lists the table of contents of the archive |
| `-x` | Extracts a file from the archive |

Example 5-1 shows how to use the `-q` option to build an archive file and the `-t` option to list its contents.

**Example 5-1** Building an Archive

```
%gcc -c foo1.c                      # creates foo1.o
%gcc -c foo2.c                      # creates foo2.o
%ar -q archive.a foo1.o foo2.o      # creates archive.a
%ar -t archive.a                    # lists contents of archive
foo1.o
foo2.o
```

Table 5-3 provides a summary of other commands that can be used to inspect and manipulate object files. Like `ar`(1), these commands are GNU based. It should be noted that `dis` is actually an alias for `objdump -d` rather than a separate command. Likewise there is no `elfdump` on Linux but there is `objdump`. It should also be noted that the functionality and flags accepted by the various commands differ between IRIX and

Linux. For more information, see the man pages for the various commands (e.g. `%man objdump`).

**Table 5-3** Additional Object File Tools

| IRIX | Linux | Function |
|---|---|---|
| `file` | `file` | Lists the general properties of the file |
| `size` | `size` | Lists the size of each section of the object file |
| `elfdump` | `readelf` | Lists the contents of an ELF object file |
| `ldd` | `ldd` | Lists the shared library dependencies |
| `nm` | `nm` | Lists the symbol table information |
| `elfdump` | `objdump` | Dump object file information contents |
| `dis` | `objdump -d` | Disassemble the source code |
| `strip` | `strip` | Remove the symbol table and relocation information |
| `c++filt` | `c++filt` | Demangle names for C++ (`nm -C`) |

# Debuggers

Debuggers on IRIX and Linux fall into two categories:

- Command line (text based) debuggers
- GUI (windowed) debuggers

On IRIX the ProDev WorkShop tools provides the `dbx` command line debugger and the CaseVision `cvd` GUI debugger. Both are able to debug programs compiled by any MIPSpro compiler and also support debugging of multi-threaded code. A second GUI debugger, TotalView is available from Etnus Corp (www.etnus.com). TotalView is also available from Etnus for Altix machines.

## Altix Command Line Debuggers

Debuggers that ship with Altix machines are provided by Intel and GNU. The Intel debugger is called `idb`. Like its GNU counterpart, `gdb,` it is a command line debugger that can attach to a running process or debug a core file. It supports debugging programs written in all of the languages supported by the Intel compilers and has been improved in the area of debugging multithreaded applications (OpenMP or pthreads). By default, it supports `dbx` commands though it can also (via option) support `gdb` commands. Table 5-4 lists some of the more commonly used commands of these debuggers.

**Table 5-4** Command Line Debugger Commonly Used Commands

| MIPSpro dbx and idb Default Command | gdb Command | Function |
|---|---|---|
| `run` | `run` | Start program |
| `continue` | `continue` | Continue stopped program |
| `attach` *pid* | `attach` *pid* | Attach to running process |
| `stop in` *function* | `break` *func* | Set breakpoint in function |
| `stop at` *line* | *break line* | Set breakpoint on line # |
| `status` | `info` | Print breakpoints |
| `delete` *N* | `delete` | Delete breakpoint |
| *print expr* | *print expr* | Print expression value |
| `step` | `step` | Single step (into functions) |
| `next` | `next` | Single step (over functions) |
| `return` | `finish` | Continue running until current function returns |
| `printregs` | `info registers` | Print register values |
| *address/Ni* | `disassemble` | Disassemble source code |
| `list` | `list` | List source code |
| `exit` | | Exit debugger |

A full set of commands supported by the `idb` and `gdb` debuggers can be found by listing their respective man pages `idb`(1) and `gdb`(1). Documentation on `gdb` is available at the GNU web site: http://www.gnu.org/software/gdb/documentation/.

## GUI Debuggers on Altix.

In addition to the previously mentioned Etnus TotalView debugger (a discussion of which is beyond the scope of this manual), there also exists a graphical front-end interface to either `gdb` (by default) or `idb` called DataDisplayDebugger or `ddd`. (For information, see http://www.gnu.org/software/ddd/.)

To invoke `ddd` running `idb` in dbx mode type, execute the following:

```
% ddd --debugger idb --dbx ./a.out
```

This creates a debugger console window where debugger commands can be typed. This also creates window panes for the source code, disassembled code, and array values. You can use the **View** menu to switch these panes on and off.

Figure 5-1 shows a typical `ddd` display.

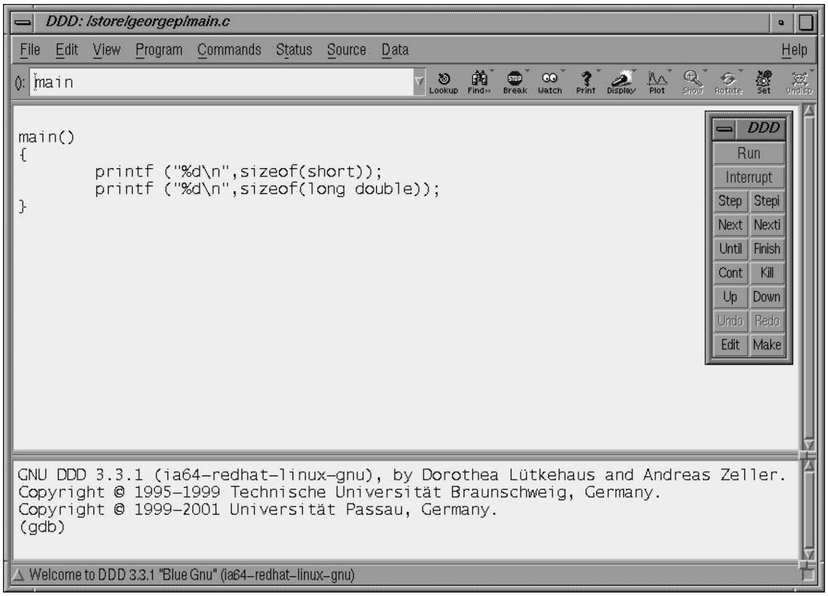


**Figure 5-1** Typical `ddd` display

Some commonly used commands from Table 5-4 are found in the **Program** pull-down menu and command.

The following site contains a thorough repository of information about `ddd`: http://www.gnu.org/manual/ddd/html_mono/ddd.html. The `ddd`(1) man page is also useful.

Another GUI debugger available for Altix is called the distributed debugging tool (DDT), available from Streamline Computing. DDT focuses on providing support for debugging parallel applications. For more information see: http://www.streamline-computing.com/softwaredivision_1.shtml.

## Timing

A variety of hardware and software support is provided for timing on IRIX and Altix systems. Understanding their implementation is critical to avoiding faulty conclusions when measuring application performance. Under IRIX systems, this support is summarized in the `timers`(5) man page. The rest of this discussion focuses on Altix systems and briefly outlines the differences between the systems.

On Altix platforms the IA64 processor has a high-resolution timer register that operates at the clock speed of the processor. This timer is available through the application register `AR.ITC`, and is commonly referred to as the itc. While providing 1 nanosecond resolution (at 1GHz), the itc registers are not synchronized across processors. Likewise the Altix Numalink hardware provides a timer that currently gives 40 nanosecond resolution. This timer is the `SN.RTC` and its value is synchronized across processors on Altix.

The basic LINUX `gettimeofday`() system call uses a pointer to a `timeval` structure containing two long integers used to return the time of day in seconds and microseconds since midnight (00:00) Coordinated Universal Time (UTC), January 1, 1970. The following example illustrates its use:

**Example 5-2** Using gettimeofday()

```
%cat td.c
#include <stdio.h>
#include <sys/time.h>

main()
{
   int i;
   struct timeval T;
   i= gettimeofday(&T,0);
```

```
   if (i==0)
      printf(“gettimeofday returned %ld seconds and %ld
microseconds\n”,T.tv_sec, T.tv_usec);

}

%icc td.c
%./a.out
gettimeofday returned 1078969017 seconds and 370026 microseconds
```

The `gettimeofday` values are updated by on every timer interrupt in the kernel. Currently this occurs at the rate of 1024 interrupts per second. If better resolution is required, variants of `clock_gettime`() can be used.

## Clock_gettime() and Clock_getres()

The `clock_gettime` function returns the current value for the specified clock (passed in by the first parameter `clock_id`). The value is returned through a pointer to a `timespec` structure consisting of two long integers containing values for seconds and nanoseconds.

Depending on the clock's resolution, it may be possible to obtain the same time value with consecutive reads of the clock. The time value may also have a higher precision then the resolution of the clock.

The resolution of any clock can be obtained by calling the `clock_getres`() function. The resolution of the clock will be returned through a pointer to a `timespec` structure.

On Altix systems the list of supported clocks differs from those on IRIX. These clocks are:

`CLOCK_REALTIME`
: The system's notion of the current time is obtained with this clock. It is currently implemented by calling `gettimeofday` and thus has the same resolution. This clock is also supported on IRIX systems.

`CLOCK_PROCESS_CPUTIME_ID`
: The processes elapsed time is obtained with this clock. It is currently implemented by reading the `SN.RTC` timer which is synchronized across nodes. As such it has submicrosecond resolution which can be obtained by the `clock_getres`() call. This clock is not supported on IRIX systems.

CLOCK_THREAD_CPUTIME_ID
: The thread's elapsed time exclusive of its parent is obtained with this clock. It is also currently implemented by reading the SN.RTC timer which is synchronized across nodes. As such it has sub microsecond resolution which can be obtained by the clock_getres() call.

  This clock is not supported on IRIX systems.

The CLOCK_SGI_CYCLE and CLOCK_SGI_FAST supported on IRIX systems are not supported on Altix system.

Example 5-3 gets the resolution of these clocks and then uses CLOCK_PROCESS_CPUTIME_ID to time how long it took to do so.

**Example 5-3** Determining clock resolution time

```
%cat tr.c

#include <stdio.h>
#include <time.h>

main()
{
   int i;
   struct timespec N;

   i = clock_getres(CLOCK_REALTIME, &N);
   if (i == 0)
      printf("Resolution is %ld seconds and  %lld nanoseconds for
CLOCK_REALTIME \n",N.tv_sec, N.tv_nsec);
   i = clock_getres(CLOCK_PROCESS_CPUTIME_ID, &N);
   if (i == 0)
      printf("Resolution is %ld seconds and  %lld nanoseconds for
CLOCK_PROCESS_CPUTIME_ID\n",N.tv_sec, N.tv_nsec);
   i = clock_getres(CLOCK_THREAD_CPUTIME_ID, &N);
   if (i == 0)
      printf("Resolution is %ld seconds and  %lld nanoseconds for
CLOCK_THREAD_CPU_TIME_ID\n",N.tv_sec, N.tv_nsec);

   i = clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &N);
   if (i == 0)
      printf("Elapsed time is %ld seconds and  %lld nanoseconds
\n",N.tv_sec, N.tv_nsec);

}

%icc tr.c -lrt
```

```
%./a.out
Resolution is 0 seconds and  976562 nanoseconds for CLOCK_REALTIME
Resolution is 0 seconds and  40 nanoseconds for
CLOCK_PROCESS_CPUTIME_ID
Resolution is 0 seconds and  40 nanoseconds for
CLOCK_THREAD_CPU_TIME_ID
Elapsed time is 0 seconds and  9125600 nanoseconds
```

As mentioned before, the `CLOCK_REALTIME` clock calls `gettimeofday` and has a resolution of 1/1024 seconds. The other two clocks provide much better resolution.

## MPI Timing Routines

MPI applications can take advantage of two portable timing routines provided with the library calls `MPI_Wtime`() and `MPI_Wtick`(). Both calls return double precision floating point numbers which represent the time and resolution in seconds respectively. They also read the `SN.RTC` and have submicrosecond resolution that is synchronized across nodes.

The following is a Fortran example that uses these timing routines.

**Example 5-4** Using MPI Timing Routines

```
%cat m.f
          PROGRAM M
          INCLUDE “mpif.h”
          DOUBLE PRECISION TME1
          DOUBLE PRECISION TME2
          DOUBLE PRECISION ELAPSED
          DOUBLE PRECISION RES
          INTEGER error

          CALL MPI_INIT(error)

          TME1=MPI_WTIME()
          RES=MPI_WTICK()
          PRINT *,”RESOLUTION IS “,RES
          PRINT *,”TIME1 IS “,TME1

          TME2=MPI_WTIME()
          PRINT *,”TIME2 IS “,TME2
          ELAPSED = TME2-TME1
          PRINT *,”ELAPSED TIME IS “,ELAPSED
          CALL MPI_FINALIZE(error)
          END
```

```
%ifort m.f -o m -lmpi
%mpirun -np 1 m
 RESOLUTION IS   4.000000000000000E-008
 TIME1 IS    207612.193793240
 TIME2 IS    207612.196069960
 ELAPSED TIME IS   2.276720013469458E-003
```

## Performance Analysis Tools

Performance analysis tools typically work in two phases. First, the application is run and performance data is collected. Typically, this data is created by one of three methods:

- Periodically interrupting a running application and capturing the program counter (PC-sampling) or the entire stack frame (Call-stack Sampling).
- Instrumenting the executable program to generate performance data as it executes certain (or all) parts of the program.
- Using hardware to detect and track certain events.

On Itanium based systems the third category is particularly important. The Itanium 2 Performance Monitoring Unit (PMU) defines over four hundred different events that can be measured in four 48-bit counters. The different types of events that can be measured fall into the following categories:

- Basic Events (Clock cycles, Retired instructions)
- Instruction Dispersal Events
- (18 events; `FP_OPS_RETIRED`, `FP_FLUSH_TO_ZERO`)
- Instruction Execution Events
- Stall Events
- Branch Events
- Memory Hierarchy
- System Events
- TLB Events
- System Bus Events
- Register Stack Engine Events

In the second phase, the collected data is analyzed and presented to the user. As with debuggers, the presentation of performance analysis tools is classified into two categories:

- Command Line (text based)
- GUI (windowed)

On IRIX, tools such as `perfex`(1), `SpeedShop`(1) and `prof` fall into the first category while `cvperf` (in ProDev WorkShop) falls in the latter.

## Performance Tools on Altix

Performance Tools on Altix are available from both Intel and the open source community (including SGI contributions). The following sections briefly document the following tools:

- `VTune` (Intel)
- `gprof` (GNU)
- `pfmon` (HP labs)
- `profile.pl` (SGI)
- `histx` (SGI)

## VTune

`VTune` (see: http://www.intel.com/software/products/vtune/vlin/) provides call stack sampling as well as comprehensive support for event based sampling of the Itanium PMU. Two versions of the tool are available. The first requires that the collected data be copied from the Altix to Windows based machine where the analysis takes place under a GUI framework. The second is a command line tool natively hosted on the Itanium system where the data was collected.

For additional information, see http://ssales.corp.sgi.com/products/servers/altix350/intelfaq.html and scroll down to find the comparison chart for VTune 7.1 versus VTune 2.0.

## gprof

The `gprof` tool requires that the application being analyzed be compiled with the `-pg` option of `gcc`. When run, the resulting program creates a `gmon.out` file which contains information that can be used to generate three types of reports by the command line based `gprof` tool:

Flat Profile
: Shows how much time your program spent in each function, and how many times that function was called.

Call Graph
: Shows, for each function, which functions called it, which other functions it called, and how many times.

Annotated Source
: Shows how many times each line of the programs source code was executed.

For further information, see the `gprof` man page (`%man gprof`).

## pfmon

The `pfmon` tool uses the Itanium Performance Monitoring Unit (PMU) to count and sample during runs made on unmodified binaries. It can function on a per-process basis or take a system-wide view on a dedicated CPU or a set of CPUs. It also can monitor events at the user level or at the system level.

The `-l` option to `pfmon` lists the (currently 475) supported events. These event names can then be used as arguments to the `-e` option which specifies which events to monitor. For example, executing the following command will monitor four different events:

```
%pfmon -ecpu_cycles,ia64_inst_retired_this,nops_retired, \
back_end_bubble_all a.out
```

Note: that there is no space between the `-e` option and the name of the first event or between the commas.

This is recommended as the first step in using `pfmon` to count cycles, instructions, NOPs and back-end stall cycles.

More information about `pfmon` can be found in its man page (%`man pfmon`) and a user guide normally installed under
`/usr/share/doc/pfmon-2.0/pfmon_usersguide.txt.`

## profile.pl

`profile.pl` is a Perl script interface to `pfmon`. It uses `dplace` to bind the application to specific processors and invoke other Perl scripts to generate a readable report. It requires that the application contain symbol table information (i.e., not be stripped). Table 5-5 shows some commonly used options to `profile.pl`.

**Table 5-5** profile.pl Flags

| Option | Meaning |
|---|---|
| `-C`*processor_list* | Used by `dplace` to bind processes to processors |
| `-E`*event* | `pfmon` event name (`CPU_CYCLES` is the default) |
| `-N`*number* | Controls how often sampling is done |
| `-O`*filename* | Puts analysis file into *filename* (`profile.out` is the default) |
| `-K` | Keep each CPU sample file and produce a separate report for each CPU |

For more information see the `profile.pl`(1), `analyze.pl`(1), and `makemap.pl(`1) man pages.

## SGI Histx

SGI Histx is a performance analysis tool designed to complement `pfmon`. The software is designed to run on Altix systems only. Used internally by SGI developers and benchmarkers, the product is offered as a service to SGI customers with a no fee end-user proprietary license via the SGI Download Cool Software (DCS) Web site. Customers wishing to use SGI Histx should be aware that there is no support planned for this product and customers who use it accept it "as is".

Histx consists of a group of tools:

First there are three data collection programs:

`libfpm`
: This tool resembles the `perfex` tool on IRIX. It supports individual threads and MPI processes reporting counts of specified events for the entire run of the program.

`samppm`
: Similar to `libfpm`, it tracks counts of events as a function of time. The binary output file is then processed by `dumppm` into a report.

`histx` Provides PC (or more accurately instruction pointer, or ip) sampling and call stack sampling

Then there are three filters for performance data postprocessing and display:

`dumppm` Formats `samppm` data into a report.

`iprep` Formats `histx` PC (ip) sampling data into a report

`csrep` Formats `histx` call stack sampling data into a report that resembles an IRIX SpeedShop "butterfly" report.

The `histx` command does not have a man page; however, typing the command by itself (or `%histx -h`) will print relevant options. For example:

```
%histx
usage: histx [-b width] [-f] [-e source] [-h] [-k] -o file [-s type]
[-t signo] command args...
  -b  specify bin bits when using ip sampling: 16,32 or 64 (default:
16)
  -e  specify event source (default: timer@1)
  -f  follow fork (default: off)
  -h  this message (command not run)
  -k  also count kernel events for pm source (default: off)
  -l  include line level counts in ip sampling report (default: off)
  -o  send output to file.<prog>.<pid> (REQUIRED)
  -s  type of sampling (default: ip)
  -t  `toggle' signal number (default: none)

Event sources:
   timer@N       profiling timer events. A sample is recorded
                 every N ticks.
   pm:<event>@N  performance monitor events.  A sample is
                 recorded whenever the number of occurrences of
                 <event> is N larger than the number of occurrences
                 at the time of the previous sample.
   dlatM@N       A sample is recorded whenever the number of
                 loads whose latency exceeded M cycles is N larger
                 than the number at the time of the previous
                 sample. M must be a power of 2 between 4 and
                 4096

Types of sampling:
   ip              Sample instruction pointer
   callstack[N]    Sample callstack.  N, if given, specifies
                   the maximum callstack depth (default: 8)
```

```
Notes:
   A list of valid performance monitor <event>s can be found
   in Intel manuals.
   `command' must not be compiled using the `-p' compiler flag
   One tick is about 0.977 milliseconds
```

Thus

```
%histx -e timer@1 -o out ./a.out
```

will generate the output file `out.a.out.`*XXXX* (where *XXXX* is the process id) which provides the number of timer ticks for each function in the `a.out` file.

# Message Passing on IRIX and Linux

This chapter describes the differences in support for the Message Passing Toolkit (MPT) on IRIX and Linux systems. For general information on using MPT under Linux, see the *Message Passing Toolkit (MPT) User's Guide.*

## Compiling MPI Programs on Linux

The compile and link syntax for MPI programs is similar on IRIX and Linux systems. See the `mpi`(1) man page for more specific information and compiler command syntax.

## SHMEM Program Launch

On Linux, SHMEM programs are launched using the `mpirun` command on one or more Altix partitions or hosts. On IRIX, `SHMEM` programs were started by setting the NPES environment variable and running the executable program directly.

On IRIX systems, `SHMEM` processes start via a fork in the `start_pes`() function call. On Linux systems, the `SHMEM` processes are MPI processes that are forked prior to entry of the main program.

See the `shmem`(3) man page for more information.

## NUMA Placement

On IRIX systems, MPI will automatically distribute the program's processes in a reasonable way across the CPUs within the system or a cpuset using the Memory Management Control Interface (MMCI) interface provided in IRIX.

On Linux systems, MPI will distribute the processes from CPU 0 to N-1 on the system or within a cpuset when exclusive execution mode is selected. MPI's exclusive execution mode can be activated in a couple ways. One way is for the user to set the `MPI_DSM_DISTRIBUTE` environment variable. Alternatively, when LSF launches an MPI program into cpusets that are dedicated to this program, it will set exclusive execution mode in the launched MPI program.

See the `mpi`(1) man page for more information.

## dplace Command

The `dplace` command can also be used to specify NUMA placement of MPI, OpenMP, and other parallel programs on IRIX and Linux systems. The `dplace` command syntax is substantially revised on Linux systems.

On IRIX, MPI programs were started this way using dplace:

```
%mpirun -np 4 dplace -place placement_file a.out
```

On Linux, MPI programs were started this way using dplace:

```
%mpirun -np 4 dplace -s1 a.out
```

See the `dplace`(1) man page for more information.

## Performance Tuning Tools

On IRIX systems, SGI SpeedShop and `perfex` are available for monitoring performance of parallel programs. On Linux, the `profile.pl` tool is available for this purpose. You can run `profile.pl` with MPI in this way:

```
%mpirun -np 4 profile.pl -s1 a.out
```

## MPT Release Documentation

On IRIX systems, the `relnotes` command could be used to read MPT release notes. On Linux systems, you can find the name of the file containing release notes information using this command:

```
%rpm -ql sgi-mpt | grep relnotes
```

## Performance Impact of Partitioning

On IRIX systems, the optimized MPI data transfer methods were never implemented for MPI programs that are run across multiple partitions. On Linux systems, the latency and bandwidth of MPI communication is the same whether you are communication inside a single host or between partitions. The only effect on performance will arise from the number of hardware routers that lie in the path between the CPUs that are running the MPI processes.

## Software Modules Differences

You can install SCSL, MPT, and other library packages in alternate locations using Software Modules on Linux. However, the rich set of compiler wrapper scripts (for example. `cc` and `f77`) do not exist on Linux. Therefore, you need to specify the `-I` and `-L` options when compiling or linking with libraries that are installed in alternate locations.

## System-Specific MPT Features

The following table summarizes the MPT features that are available on IRIX only and the MPT features that are available on Linux only.

**Table 6-1** System-Specific MPT features

| IRIX only | Linux only |
|---|---|
| Support for 48p x 128p clusters | Optimized MPI send/ across partitions |
| Support for up to 512p single hosts | MPI one-sided across partitions |
| Support for checkpoint and restart (CPR) | SHMEM across partitions |
| Fortran 90 compile-time MPI interface checking | |
| MPI-2 capabilities | |
| Support for MPI_Comm_spawn and MPI_Comm_spawn_multiple | |
| Thread safety | |
| USEM MPI Fortran 90 statement support | |

# POSIX Threads (pthreads) Implementations

A thread is a sequence of instructions to be executed within a program. Normal UNIX processes consist of a single thread of execution, along with system resources (such as open files) and a virtual address space. The overhead associated with process creation, destruction and context switching led to the development of various "lightweight process" and threading libraries. They sought to minimize this overhead by having the threads share various resources and thus the operating system would have less to do on thread creation etc.

Historically, various vendors have implemented their own proprietary versions of lightweight processes and threads. For example IRIX implemented shared lightweight processes or sprocs. These implementations differed substantially from each other making it difficult for programmers to develop portable applications.

In 1995 the IEEE provided a standardized thread based programming interface, POSIX 1003.1c (also known as ISO/IEC 9945-1:1996), referred to as POSIX threads or P-threads. The standard provides a variety of application programming interfaces that fall into three categories:

- Thread creation and destruction
- Thread synchronization and resource locking
- Thread management and scheduling

This chapter outlines differences between the Pthreads implementations on IRIX 6.5 and the latest version of ProPack. It must be noted that the Linux information is highly dependent on the version of the kernel and threading library being supported. The ProPack 2.4 release and glibc 2.2.4 supported the LinuxThreads library. The ProPack 3.0 release and glibc 2.3+ support the Natice Posix Thread Library for Linux (NPTL).

## Implementation Differences

As of The IRIX 6.5.20 release, IRIX is Unix98 conformant and fully compliant with the POSIX 1003.1c standard. It implements an M:N threading model whereby M threads are mapped onto N kernel processes. This allows the ability to create both kernel and user level threads and to quickly switch between thousands of them. At the same time, it does complicate the implementation.

LinuxThreads (http://pauillac.inria.fr/~xleroy/linuxthreads), on the other hand, adopts a 1:1 threading model where each thread is mapped onto a kernel process. Although this, in theory, should increase switching times, the LinuxThreads designers point to the overall low switching overhead of the Linux kernel. They also point to a simplified design that performs well when most threads are blocked or when there is not a large number of runnable threads. While LinuxThreads does implement all of the APIs from the POSIX 1003.1c standard, LinuxThreads is not standard conformant in the area of signal handling.

The Native Posix Thread Library (described in http://people.redhat.com/drepper/nptl-design.pdf) provides performance improvements and increased scalability and it aims to overcome most of the deficiencies of Linux Threads while remaining as compatible as possible to the Linux Thread API. It is also a 1:1 (rather than M:N) threading model, but it corrects many of the issues with signal handling in Linux Threads and is thus much more standard conformant. Applications that rely on behavior where the LinuxThreads implementation deviates from the POSIX standard will need to be fixed. These behavior differences include the following:

- Signal handling has changed from per-thread signal handling to POSIX process signal handling.
- `getpid`() returns the same value in all threads.
- Thread handlers registered with `pthread_atfork` are not run if `vfork`() is used.
- There is no manager thread.

If an application does not work properly with NPTL, it can be run using the old LinuxThreads implementation by setting the following environment variable:

`LD_ASSUME_KERNEL=`*kernel-version*

The following versions are available:

2.4.19 -- Linuxthreads with floating stacks

Note that software using `errno`, `h_errno`, and `_res` must `#include` the appropriate header file (`errno.h`, `netdb.h`, and `resolv.h` respectively) before they are used. However, `LD_ASSUME_KERNEL=2.4.19` can be used as a workaround until the software can be fixed.

## Differences in Cancellation

Cancellation is the mechanism by which a thread can send a request to terminate the execution of another thread. Depending on its settings the target thread can then either ignore the request, honor it immediately, or defer it till it reaches a cancellation point. Cancellation points are those points in the program execution where a test for pending cancellation requests is performed and cancellation is executed if positive.

Under IRIX the following functions are cancellation points:

```
accept(2)
aio_suspend(3)
close(2)
connect(2)
creat(2)
fcntl(2)
fsync(2)
getmsg(2)
getpmsg(2)
lockf(3C)
mq_receive
mq_send
msgrcv(2)
msgsnd(2)
msync(2)
nanosleep(2)
open(2)
pause(2)
poll(2)
pread(2)
pthread_cond_timedwait(3P)
pthread_cond_wait(3P)
pthread_join(3P)
pthread_testcancel(3P)
putmsg(2)
```

```
putpmsg(2)
pwrite(2)
read(2)
readv(2)
recv(2)
recvfrom(2)
recvmsg(2)
select(2)
sem_wait
semop(2)
send(2)
sendmsgsendto(2)
sigpause(2)
sigsuspend(2)
sigtimedwait(3)
sigwait(3)
sigwaitinfo(3)
sleep(3C)
system(3S)
tcdrain(3t)
usleep(3C)
wait(2)
wait3(2)
waitid(2)
waitpid(2)
write(2)
writev(2)
```

In contrast the following are cancellation points under Linux:

```
pthread_join(3)
pthread_cond_wait(3)
pthread_cond_timedwait(3)
pthread_testcancel(3)
sem_wait(3)
sigwait(3)
```

In particular note that no system call is a cancellation point under Linux. In contrast, under IRIX the system call wrapper checks the caller and enables and disables cancellation around the particular system call.

For more information see the following man pages on IRIX and Linux: `pthread_cancel`(3P), `pthread_setcancelstate`(3P)

## Differences in Mutex Implementations

A Mutex (or mutual exclusion point) controls whether threads can execute a critical region of code or modify a shared variable. They are a primary means of thread synchronization under Pthreads.

A mutex variable acts like a "lock" protecting access to a shared resource, such as shared memory or file descriptors. Only one thread can lock (or own) a mutex variable at any given time. If several threads try to lock a mutex, only one thread will succeed. The other threads will not be granted the mutex until the owner releases it.

A mutex has attributes that control its behavior. Under IRIX the function `pthread_mutexattr_settype`() defines the type of mutex. The type value may be one of `PTHREAD_MUTEX_NORMAL`, `PTHREAD_MUTEX_ERRORCHECK`, `PTHREAD_MUTEX_RECURSIVE`, `PTHREAD_MUTEX_SPINBLOCK_NP`, or `PTHREAD_MUTEX_DEFAULT`.

LinuxThreads supports only one mutex attribute: the mutex kind, which is either `PTHREAD_MUTEX_FAST_NP` for fast mutexes, `PTHREAD_MUTEX_RECURSIVE_NP` for recursive mutexes, or `PTHREAD_MUTEX_ERRORCHECK_NP` for error checking. mutexes. In all cases the NP suffix refers to "Non Portable" extensions to the Posix standard.

IRIX also implements a process-shared attribute (`PTHREAD_PROCESS_SHARED`) to permit a mutex to be operated upon by any thread that has access to the memory where the mutex is allocated, even if the mutex is allocated in memory that is shared by multiple processes. If the process-shared attribute is `PTHREAD_PROCESS_PRIVATE`, the mutex will only be operated upon by threads created within the same process as the thread that initialized the mutex; if threads of differing processes attempt to operate on such a mutex, the behavior is undefined. The default value of the attribute is `PTHREAD_PROCESS_PRIVATE`. For more information on IRIX (see: `pthread_mutexattr_setpshared`(3P)). This feature is not implemented under LinuxThreads (glibc 2.2.x) but is supported by NPTL (glibc 2.3+)

It should be pointed out that both IRIX and Linux support optimized atomic operations that are much faster than the following code sequence:

```
pthread_mutex_lock ( &count_mutex );
   count++;
pthread_mutex_unlock ( &count_mutex );
```

On IRIX __fetch_and_add while under Linux __sync_fetch_and_add (gcc) or _InterlockedIncrement (Intel compiler) would be much faster.

## Condition Variables

Condition variables allow threads to suspend execution until some condition is satisfied. Functions are provided to wait on a condition variable and to wake up threads that a waiting on the condition variable.

The type of condition variable used is determined by the attribute structure attr passed with the call to pthread_cond_init(). On IRIX these attributes are set by calls to pthread_condattr_init() and the various condition variable attribute functions such as pthread_condattr_init() and pthread_condattr_setpshared(). If attr is null (or the condition variable is statically initialized) the default attributes are used. The IRIX implementation supports the process-shared attribute. If this attribute is set to PTHREAD_PROCESS_SHARED it allows a condition variable to be operated upon by any thread that has access to the memory where the condition variable is allocated, even if the condition variable is allocated in memory that is shared by multiple processes. If the process-shared attribute is PTHREAD_PROCESS_PRIVATE, the condition variable will only be operated upon by threads created within the same process as the thread that initialized the condition variable. The default value of the attribute is PTHREAD_PROCESS_PRIVATE.

The LinuxThreads implementation supports no attributes for conditions, hence the cond_attr parameter is ignored by pthead_cond_init(). Likewise pthread_condattr_init() and pthread_condattr_destroy() under LinuxThreads do nothing and are only included for compliance with the POSIX API's.

NPTL supports the process-shared attribute for condition variables.

For more information see the pthread_cond_wait(3) and pthread_condattr_init(3) man pages.

## Read-Write Locks

A read-write lock is a software object that gives one thread the right to modify some data, or multiple threads the right to read that data. The pthreads library on IRIX implements several functions for initializing and using read-write locks. For more informations see `pthread_rwlock_init`(3) `pthread_rwlock_rdlock`(3) and `pthread_rwlock_wrlock`(3).

Read-write locks are extensions to the POSIX standard and are not implemented on LinuxThreads but are supported by NPTL.

## Signals

A signal is an asynchronous notification of an event.

Each thread has a signal mask that specifies the signals it is willing to receive. This mask can be changed in a pthreads program by calling the `pthread_sigmask`() function.

As mentioned earlier in this chapter, signal handling in LinuxThreads does not conform to the POSIX standard and is thus significantly different than the IRIX implementation. NPTL, on the other hand, is standard compliant.

According to the standard, external signals are addressed to the whole process (the collection of all threads), which then delivers them to the one particular thread. However, since each thread is actually a kernel process with its own process ID (PID) in LinuxThreads, external signals are always directed to one particular thread. If, for instance, another thread is blocked in `sigwait` on that signal, it will not be restarted. NPTL overcomes this by performing signal-handling for multi-threaded processes in the kernel. Signals sent to the process are now delivered to one of the available threads.

The LinuxThreads implementation of sigwait installs dummy signal handlers for the signals in set for the duration of the wait. Since signal handlers are shared between all threads, other threads must not attach their own signal handlers to these signals, or alternatively they should all block these signals.

Another difference between the implementations is that IRIX uses `SIGPTRESCHED` and `SIGPTINTR` for scheduling and cancellation whereas LinuxThreads uses `SIGRTMIN` and `SIGRTMIN+1`. NPTL uses `SIGRTMIN`

# Scheduling Pthreads

Pthreads are scheduled by their scope, policy and priority. These variables are set initially when the thread is created though policy and priority can also be modified at runtime by the `pthread_setschedparam`() function.

## Scope

IRIX supports three different contention scopes. System and bound scope threads are scheduled by the IRIX kernel, and compete with all other threads on the system. System scope threads are suitable for real-time programming and may only be created by privileged users, whereas bound scope threads are not suitable for real-time programming and do not require special privileges to create. Process scope threads are scheduled by the Pthreads library, and compete with one another for process timeslices. By default Pthreads are created with process scope.

The only scope supported by LinuxThreads is the system scope.

## Policy

IRIX supports the following policies:

- `SCHED_RR` (default; round robin scheduling)
- `SCHED_FIFO` (first in first out)
- `SCHED_TS` (time sharing same as `SCHED_RR`)
- `SCHED_OTHER` (same as `SCHED_RR`)

LinuxThreads supports these policies

- `SCHED_OTHER` (regular non-realtime scheduling)
- `SCHED_FIFO` (realtime, first-in first out)
- `SCHED_RR` (realtime, round robin)

### Priority

IRIX supports priorities between 0-255. The range on LinuxThreads is 1-99. Larger numbers represent higher priorities on both implementations.

## Environment Variables

IRIX supports the `PT_CORE` and `PT_SPINS` environment variables.PT_CORE permits a core file to be generated in certain situations which are otherwise not permitted by the Pthreads library, but should generally not be used unless debugging an application. PT_SPINS determines how many times a lock is tried before sleeping.

For more information see the IRIX `pthreads`(5) man pages. Neither are supported on Linux.

## Summary of Differences in Supported Features

A chart that illustrates various pthreads features that are supported by different variants of Unix can be found at: http://www.tldp.org/FAQ/Threads-FAQ/OSsCompared.html

Table 7-1 reproduces a portion of this chart and includes what is supported on IRIX and ProPack 3.0 (NPTL) and ProPack 2.4 (LinuxThreads) respectively.

**Table 7-1** IRIX 6.5 vs. Linux Pthread Feature Comparison

| Feature | IRIX | NPTL | Linux Threads |
|---|---|---|---|
| User(U)/Kernel(K)-space | K&U | K | K |
| Cancellations | Yes | Yes | Yes |
| Priority Scheduling | Yes | Yes | Yes |
| Priority Inversion Handling [A] | Yes | Yes | No |
| Mutex Attributes | Yes | Yes | Yes |
| Shared and Private Mutexes [B] | Yes | Yes | No |
| Thread Attributes | Yes | Yes | Yes |
| Synchronization | Yes | Yes | Yes |
| Stack Size Control | Yes | Yes | No |
| Base Address Control | Yes | No [1] | No [1] |
| Detached Threads | Yes | Yes | Yes |
| Joinable Threads | Yes | Yes | Yes |
| Per-Thread Data Handling Function | Yes | Yes | Yes |
| Per-Thread Signal Handling | Yes | Yes | Yes |
| Condition Variables | Yes | Yes | Yes |
| Semaphores | Yes | Yes | No |
| Thread ID Comparison | Yes | Yes | Yes |
| Call-Once Functions | Yes | Yes | Yes |
| Thread Suspension | No [2] | Yes | Yes |

**Table 7-1** IRIX 6.5 vs. Linux Pthread Feature Comparison **(continued)**

| **Feature** | **IRIX** | **NPTL** | **Linux Threads** |
|---|---|---|---|
| Specifying Concurrency [C] | Yes [3] | Yes | No |
| Reader/Writer Share Locking | Yes | Yes | No |
| Processor-specific Thread Allocation [D] | Yes [4] | Yes | No |
| Fork All Threads [E] | No [5] | No | No |
| Fork Calling Thread Only | Yes | Yes | Yes |

Feature Definitions:
[A] As threads get blocked on I/O, provide a temporary reprioritization of threads.
[B] Having separate spaces for mutexes
[C] The ability to identify which threads will be multiprocessed.
[D] The ability to designate a specific thread to a specific processor.
[E] A flag which forces all thread-creation calls to be forks with shared memory.

Notes:
[1] Using cpusets or `dplace` could accomplish much the same thing
[2] Only the whole process.
[3] You can specify how much user-level threads you will use at once. The number of kernel-level threads (i.e. concurrency level) is then determined as min([max number of threads to use],[number of available processors]).
[4]Via `pthread_setrunon_np`(3P).
[5] Available through the IRIX-specific `sproc`() call. However, it should be noted that sproc's and pthreads are not compatible under IRIX and cannot be intermixed.

# Miscellaneous Porting Concerns

This chapter provides a list of issues that you may need to address when porting an application from an IRIX to a Linux system.

## I/O Controls

The IRIX `syssgi`(2) system call is not available on Linux systems. In some cases, you can replace the functionality of a `syssgi` call with the `sysctl`() function.

For example, the following IRIX code will need to be modified:

```
syssgi(SGI_CELL, SGI_GET_CLUSTER_CONFIG, &clconfig)
```

On Linux, the following code provides the same functionality:

```
if(cis_syssgi(SGI_CELL, SGI_GET_CLUSTER_CONFIG,
  PTR_TO_U64(&clconfig), PTR_TO_U64(NULL),
  PTR_TO_U64(NULL), PTR_TO_U64(NULL),
  PTR_TO_U64(NULL), PTR_TO_U64(NULL)))
```

Where `cis_syssgi` is the following routine:

```
int
cis_syssgi(int64_t cmd, int64_t arg1, int64_t arg2, int64_t arg3,
                int64_t arg4, int64_t arg5, int64_t arg6, int64_t arg7)
                {
                int r;
                uint64_t    args[8];
                int name[] = {CTL_KERN, CTL_SYSSGI, 1};

                args[0] = cmd; args[1] = arg1; args[2] = arg2; args[3]
                = arg3;
                args[4] = arg4; args[5] = arg5; args[6] = arg6; args[7]
                = arg7;

                        r = sysctl(name, 3, NULL, NULL, args,
                        sizeof(args));

                        return r;
                }
```

Some variable definitions for the above code are as follows:

```
#define SGI_CELL    1060
#define SGI_GET_CLUSTER_CONFIG 22
```

Additionally, the IRIX `sysmp`(2) call is not available on Linux systems. The following examples show some equivalent Linux functionality:

- To find number of processors, replace `sysmp(MP_NPROCS)` with: `sysconf(_SC_NPROCESSORS_CONF)`.
- To pin a process to a CPU, replace `sysmp(MP_MUSTRUN, pCpu)` with the following:

```
unsigned long cpuMask[8];
int offset, bit, ullen = sizeof( unsigned long );

memset( cpuMask, 0, ullen * 8  );
offset = pCpu / (ullen * 8);
bit = pCpu % (ullen * 8);
cpuMask[offset] = ((unsigned long)1) << bit;
return syscall( __NR_sched_setaffinity, getpid(), ullen * 8, cpuMask
);
```

- To find the number of nodes replace `sysmp(MP_NUMNODES)` with a function:

```
int getNumNodes() {
int nodeCount = 0;
int goOn=1;
struct stat statData;
char path[128];

do
{
  snprintf( path, 128, “/proc/sgi_sn/node%d”, nodeCount );
  if( stat( path, &statData ) == 0 ) nodeCount++;
  else goOn = 0;
} while( goOn );

  if( nodeCount == 0 ) nodeCount == -1;
  return nodeCount;
}
```

Additionally, in some circumstances you may need to replace `syssgi`(2) calls with `ioctl`(2) calls.

## ATT Korn Shell vs. Public Domain Korn Shell

If you are porting code from the ATT Korn shell on IRIX to a public domain Korn shell on Linux, you may need to modify your scripts.

A common procedure in a sh/ksh script is in the following format:

```
1 Some-command | while read a b c; do
2        [set var foo to something]
3 done
4 echo $foo
```

In PD ksh, the while-loop will be in a subshell and when `foo` is set at line 2 it will be in the subshell. The `foo` at line 4 will not reflect the change that happened at line 2. This behavior may not be expected.

Note that this situation holds in for-loops as well.

The workaround works for PD ksh as well as the AT&T ksh that runs on Irix is to modify the script as follows:

```
1 some-command |&
```

```
2 while read -p a b c; do
3         [Set var foo to something]
4 done
5 echo $foo
```

This arrangement flips things around, pushing “some-command” into the subshell and allowing the while-loop to be in the main shell. Now any change to `foo` at line 3 will be seen at line 5.

The `|&` syntax and the matching `read -p` provide an example of ksh co-processes and reading from pipes.

While most Linux distributions use PD ksh, the AT&T ksh is also open-source and you may choose to install that in place of the PD kshd.

AT&T ksh can find it at http://www.kornshell.com/. The source is free, but the license is not GPL or BSD.

## Serial Port Devices

Serial port devices have a different naming scheme under IRIX than under Linux. A `/dev/ttyd`[*N*] device in IRIX corresponds to `/dev/ttyS`[*N-1*] in Linux.

**Table 8-1** IRIX and Linux device naming examples

| | IRIX device name | Linux device name |
|---|---|---|
| serial port 1 | `/dev/ttyd1` | `/dev/ttyS0` |
| serial port 2 | `/dev/ttyd2` | `/dev/ttyS1` |

## Security

Table 8-2 summarizes the system security available on IRIX and Altix systems.

**Table 8-2** IRIX and Linux Security Features

| | IRIX | Linux |
|---|---|---|
| Password length | yes | yes |
| Password aging | yes | yes |
| Password composition | yes | yes |
| Logging Login/Logout | yes | yes |
| Logging Failed Login | yes | yes |
| Lockout of accounts after multiple failed logins | yes | yes |
| Logging password changes | yes | requires audit trails |
| Logging access to security relevant objects | yes | requires audit trails |
| Logging security policy changes | yes | requires audit trails |
| Audit trails | yes | Coming soon |
| Displaying banners on login screens | yes | yes |
| Proper permissions set on security relevant files | yes | yes |
| Access Control Lists | yes | yes |
| Common Criteria Security CAPP certification | EAL3 | Planning underway, please contact SGI |
| Common Criteria Security LSPP certification | EAL3 for Trusted IRIX | Planning underway, please contact SGI |
| NISPOM Chapter 8 | yes | Available in SGI ProPack 3.0 for Linux with patch |
| DII-COE | yes | Planning underway, please contact SGI |

# Frequently Asked Questions

This chapter gathers frequently asked questions and provides quick answers. When possible, supplementary reference material is provided.

**Q.** I did not see any references to Java in ProPack. What should I do?

**A.** Java for IA64 Linux is available from BEA Systems. See their websites:

For Downloads see
http://commerce.bea.com/showallversions.jsp?family=WLJR.

For redistribution terms see:
http://commerce.bea.com/products/weblogicjrockit/support_services.jsp

A version is also available from Sun at: http://java.sun.com/j2se/1.4.2/download.html

**Q.** I profiled my application, and got a list of functions that I could not find documentation for. The list of functions is:

`__kmp_wait_sleep`

`__kmp_yield`

`__kmp_static_yield`

`__kmp_ia64_pause`

`__kmp_fork_call`

`__kmp_acquire_bootstrap_lock`

What are they?

**A.** These are internal functions in Intel's libguide. They are not documented.

**Q.** I want to use a “perfex like” performance analysis tool on Altix. What should I use?

**A.** Tools such as `pfmon`, `profile.pl` and `histx` fall into this category. See Chapter 5 for more information.

**Q.** How do I disassemble my binary? There is no `dis`(1).

**A.** Use `objdump -d`.

**Q.** Can I read big-endian formatted files with my Fortran program?

**A.** Yes, set the `F_UFMTENDIAN` environment variable to big.

**Q.** Is there the equivalent of the `MP_SLAVE_STACKSIZE` environment variable on Altix?

**A.** `KMP_STACKSIZE`

see http://developer.intel.com/software/products/kappro/kappro_manual.pdf (page 76)

**Q.** I have a subroutine that the Intel compiler asserts cannot be optimized at any level (-O0 through -O3 using various 7.1 versions) because of resource constraints. Is there a magic flag, like `-OPT:Olimit=0` in the MIPSpro compilers, that I can use to get some optimization out of the compiler for this routine?

**A.** The `-override_limits` flag sometimes helps in these cases.

**Q.** The following fragment of Fortran code compiled with version 7.1 of the Intel compilers (and with MIPSpro) but does not with version 8 of the Intel compilers. What is the matter?

```
subroutine foo(x, n)
implicit none
real x(n)
integer n
end
```

**A.** Version 8 sees the `implicit none` and determines that `n` is of an undefined type. Reversing the declarations of `x` and `n` will compile.

**Q.** What tools are available that will help me port my 32-bit application to 64-bits?

**A.** The compiler helps the most in this regard. Pay special attention to the warnings generated. A script that may help in this is the following:

```
#!/usr/bin/env python
#
# Copyright (c) 2004 Hewlett-Packard Development Company, L.P.
#       David Mosberger <davidm@hpl.hp.com>
#
# Scan standard input for GCC warning messages that are likely to
# source of real 64-bit problems.  In particular, see whether there
# are any implicitly declared functions whose return values are later
# intepreted as pointers.  Those are almost guaranteed to cause
# crashes.
#
import re
import sys

implicit_pattern = re.compile("([^:]*):(\d+): warning: implicit declaration "
                               + "of function `([^']*)'")
pointer_pattern = re.compile("([^:]*):(\d+): warning: "
                              + "(assignment"
                              + "|initialization"
                              + "|return"
                              + "|passing arg \d+ of `[^']*'"
                              + "|passing arg \d+ of pointer to function"
                              + ") makes pointer from integer without a cast")
while True:
     line = sys.stdin.readline()
     if line == '':
         break
     m = implicit_pattern.match(line)
     if m:
         last_implicit_filename = m.group(1)
         last_implicit_linenum = int(m.group(2))
         last_implicit_func = m.group(3)
     else:
        m = pointer_pattern.match(line)
         if m:
             pointer_filename = m.group(1)
             pointer_linenum = int(m.group(2))
             if (last_implicit_filename == pointer_filename
                 and last_implicit_linenum == pointer_linenum):
                 print "Function `%s' implicitly converted to pointer at " \
```

```
“%s:%d” % (last_implicit_func, last_implicit_filename,
         last_implicit_linenum)
```

This Python script scans the output of `gcc -Wall -O` for warnings that are almost guaranteed to cause crashes on ia64. This won’t help much for applications that are hopelessly 64-bit-dirty, but it will help those applications that are basically 64-bit clean, save for some silly oversights (like missing header file includes).

Here is an example:

```
$ check-implicit-pointer-functions < ./log
Function `strdup’ implicitly converted to pointer at e-pilot-util.c:42
Function `e_path_to_physical’ implicitly converted to pointer at
mail-importer.c:98
```

**Q.** Does the `cpio` command on Altix support the `-K` IRIX feature?

**A.** The IRIX `cpio -K` option turns on use of the extended format and is required for files larger than 2 Gigabytes. IRIX `cpio` gives a warning message whenever a non-standard `cpio` file is written. This option is not available on Altix. As an alternative there, use the GNU `tar` command which can archive files up to 64 gigabytes.

# Application Programming Interface (API) Differences: libc

This chapter summarizes the library routines that are available on IRIX but missing on Linux.

This chapter covers only the routines in the Standard C libraries (`libc`). Issues surrounding porting of MPI libraries are documented in Chapter 6, and POSIX threading libraries (`libpthread`) are documented in Chapter 7. Other libraries will be added in subsequent releases of this manual.

IRIX has a variety of library calls in libc that are either missing in the Linux `libc` or in a different library. The following attempts to group the differences into categories.

Arena memory allocations routines:

- `acreate`
- `adelete`
- `afree`
- `amallinfo`
- `amalloc`
- `amallocblksize`
- `amallopt`
- `amemalign`
- `arealloc`
- `arecalloc`
- `usdetach`
- `usadd`
- `usinit`

- `uscalloc`
- `usfree`
- `usmallinfo`
- `usmalloc`
- `usmallopt`
- `usrealloc`

Selected asynchronous I/O functions:

- `aio_hold`
- `aio_hold64`
- `aio_sgi_init`
- `aio_sgi_init64`

Selected Time conversion functions:

- `ascftime`
- `cftime`

BSD compatibility routines:

- `BSDalphasort`
- `BSDchown`
- `BSDclosedir`
- `BSDdup2`
- `BSDfchown`
- `BSDgetgroups`
- `BSD_getime`
- `BSDgetpgrp`
- `BSDgettimeofday`
- `BSDinitgroups`
- `BSDlongjmp`

- BSDopendir
- BSDreaddir
- BSDscandir
- BSDseekdir
- BSDsetgroups
- BSDsetjmp
- BSDsetpgrp
- BSDsettimeofday
- BSDsignal
- BSDsigpause
- BSDtelldir

Capabilities related Routines:

- cap_acquire
- cap_clear
- cap_copy_ext
- cap_copy_int
- cap_dup
- cap_envl
- cap_envp
- cap_free
- cap_from_text
- cap_get_fd
- cap_get_file
- cap_get_flag
- cap_get_proc
- cap_init
- cap_set_fd

- `cap_set_file`
- `cap_set_flag`
- `cap_set_proc`
- `cap_set_proc_flags`
- `cap_size`
- `cap_surrender`
- `cap_to_text`
- `cap_value_to_text`

Library routines for dealing with creation and manipulation of CLIENT handles:

- `clnt_broadcast_exp`
- `clnt_broadmulti`
- `clnt_broadmulti_exp`
- `clnt_create_vers`
- `clnt_dg_create`
- `clnt_multicast`
- `clnt_multicast_exp`
- `clnt_raw_create`
- `clnt_setbroadcastbackoff`
- `clnt_syslog`
- `clnt_tli_create`
- `clnt_tp_create`
- `clnt_vc_create`

Select routines that maintain key/content pairs in a data base:

- `dbm_clearerr64`
- `dbmclose64`
- `dbm_close64`
- `dbm_delete64`

- dbm_error64
- dbm_fetch64
- dbm_firstkey64
- dbm_forder
- dbm_forder64
- dbminit64
- dbm_open64
- dbm_store64
- delete
- delete64
- firstkey
- firstkey64
- nextkey
- nextkey64

Long double conversion routines:

- ecvtl
- fcvtl
- gcvtl
- ecvtl_r
- fcvtl_r

Networking file entry manipulation routines:

- fgethostent
- fgethostent_r
- fgetnetent
- fgetnetent_r
- fgetprojall
- fgetprojuser

- `fgetprotoent`
- `fgetprotoent_r`
- `fgetrpcent`
- `fgetrpcent_r`
- `fgetservent`
- `fgetservent_r`

Hardware Inventory entry functions:

- `getinvent`
- `setinvent`
- `endinvent`
- `scaninvent`
- `getinvent_r`
- `setinvent_r`
- `endinvent_r`

Networking configuration database entry functions:

- `getnetconfig`
- `endnetconfig`
- `getnetconfigent`
- `freenetconfigent`
- `nc_perror`
- `nc_sperror`
- `setnetpath`
- `getnetpath`
- `endnetpath`

Job limits functions:

- `killjob`

- makenewjob
- waitjob
- setwaitjobpid
- jlimit_startjob
- getjlimit
- setjlimit

Three byte integer conversion routines:

- l3tol
- ltol3

MAC label manipulator functions:

- mac_clearance_error
- mac_cleared
- mac_cleared_fl
- mac_cleared_fs
- mac_clearedlbl
- mac_cleared_pl
- mac_cleared_ps
- mac_demld
- mac_dominate
- mac_dup
- mac_equal
- mac_free
- mac_from_mint
- mac_from_msen
- mac_from_msen_mint
- mac_from_text
- mac_get_fd

- `mac_get_file`
- `mac_get_proc`
- `mac_is_moldy`
- `mac_label_devs`
- `mac_set_fd`
- `mac_set_file`
- `mac_set_moldy`
- `mac_set_proc`
- `mac_size`
- `mac_to_text`
- `mac_to_text_long`
- `mac_valid`

MINT label manipulator functions:

- `mint_dom`
- `mint_equal`
- `mint_free`
- `mint_from_mac`
- `mint_from_text`
- `mint_size`
- `mint_to_text`
- `mint_valid`

Memory Locality Domain Operations:

- `mld_create`
- `mld_create_special`
- `mldset_create`
- `mldset_create_special`
- `mldset_destroy`

- mldset_place
- process_mldlink

Message queue descriptor functions:

- mq_close
- mq_getattr
- mq_notify
- mq_open
- mq_receive
- mq_send
- mq_setattr
- mq_unlink

MSEN label manipulator functions:

- msen_dom
- msen_equal
- msen_free
- msen_from_mac
- msen_from_text
- msen_size
- msen_to_text
- msen_valid

Lightweight process creation routines:

- pcreatel
- pcreatelp
- pcreatev
- pcreateve
- pcreatevp

- `sproc`
- `sprocsp`

Process module routines:

- `pm_attach`
- `pm_create`
- `pm_create_simple`
- `pm_create_special`
- `pm_filldefault`
- `pm_getall`
- `pm_getdefault`
- `pm_getstat`
- `pm_setdefault`
- `pm_setpagesize`

Functions to execute a file on a remote call

- `rexecl`
- `rexecle`
- `rexeclp`
- `rexecv`
- `rexecve`
- `rexecvp`

Functions to send a signal to a process or a group of processes:

- `sig2str`
- `sigflag`
- `sigpoll`
- `sigsend`
- `sigsendset`

- `sigwaitrt`

System routines:

- `sysget`
- `sysid`
- `sysmips`
- `sysmp`
- `syssgi`

Trusted networking functions:

- `tsix_get_mac`
- `tsix_get_solabel`
- `tsix_get_uid`
- `tsix_off`
- `tsix_on`
- `tsix_recvfrom_mac`
- `tsix_sendto_mac`
- `tsix_set_mac`
- `tsix_set_mac_byrhost`
- `tsix_set_solabel`
- `tsix_set_uid`

Universal Unique Identifier Functions:

- `uuid_create`
- `uuid_create_nil`
- `uuid_equal`
- `uuid_from_string`
- `uuid_hash`
- `uuid_hash64`

- `uuid_is_nil`
- `uuid_to_string`

Selected wide character type (wchar_t) string operations and type transformations:

- `isnumber`
- `isphonogram`
- `isideogram`
- `isenglish`
- `isspecial`
- `issubdir`
- `iswascii`
- `wcstok_r`
- `wscat`
- `wschr`
- `wscmp`
- `wscpy`
- `wscspn`
- `wslen`
- `wsncat`
- `wsncmp`
- `wsncpy`
- `wspbrk`
- `wsrchr`
- `wsspn`
- `wstostr`

# Index