HP-UX 64-bit Porting and Transition Guide

HP 9000 Computers

5966-9887

June 1998

Legal Notices

HEWLETT-PACKARD COMPANY
3000 Hanover St.
Palo Alto, CA 94304
U.S.A.

Contents

Contents

Contents

Contents

Preface

The *HP-UX 64-bit Porting and Transition Guide* is a tool to help you transition to the HP-UX 64-bit platform. This manual describes the changes you need to make to compile, link, and run programs in 64-bit mode.

This *Guide* covers the following topics:

- Chapter 1, “Overview,” provides an overview of features available in the 64-bit computing environment.
- Chapter 2, “Summary of Changes,” provides a summary of changes to compiler products to support 64-bit development.
- Chapter 3, “HP-UX 64-bit Porting Concepts,” describes the 64-bit data model and how it impacts porting.
- Chapter 4, “Transitioning C and aC++ Programs to 64-bit Mode,” provides steps for transitioning C and aC++ programs to 64-bit mode.
- Chapter 5, “Writing Portable Code,” provides information on using industry standard and HP-provided portability features.

Related Documentation

For more information on programming in the HP-UX 64-bit environment, refer to the following documentation:

- *HP-UX Applications Interoperability White Paper,* URL:http://www.software.hp.com/STK/
- *HP C Online Reference* (`cc +help`)
- *HP aC++ Online Programmer's Guide* (`aCC +help`)
- *HP Fortran 90 Programmer's Reference* and *HP Fortran 90 Programmer's Notes*
- *HP-UX Linker and Libraries Online User's Guide* (`ld +help`)
- *PA-RISC 2.0 Architecture* by Gerry Kane (Prentice-Hall, ISBN 0-13-182734-0)
- *Assembler Reference Manual (92432-90012)*

Finding this Manual

The *HP-UX 64-bit Porting and Transition Guide* along with many other HP-UX books is available on the HP-UX 11.0 Instant Information CD-ROM and on the World-Wide Web. You can find this guide in the following locations:

- http://docs.hp.com/hpux/development, the HP-UX Systems Information & Documentation site
- http://www.software.hp.com/STK/, the HP-UX 11.0 Software Transition Kit

Printing History

New editions of this manual will incorporate all material updated since the previous edition.

The software version is the version level of the software product at the time the manual was issued. Many product updates and fixes do not require manual changes and, conversely, manual corrections may be done without accompanying product changes. Therefore, do not expect a one-to-one correspondence between product updates and manual updates.

Edition	Date	Part Number	HP-UX Software Version
First Edition	November 1997	5966-9844	HP-UX 11.0
Second Edition	June 1998	5966-9887	HP-UX 11.0 June 1998 Extension Pack

You can send any suggestions for improvements in this manual to:

Languages Information Engineering Manager
Hewlett-Packard Company
Mailstop 42UD
11000 Wolfe Road
Cupertino, CA 95014-9804

1 Overview

The HP-UX 11.0 release offers several features to enable the full power of 64-bit computing. It includes tools to help you transition applications from 32-bit to 64-bit mode. This chapter covers the following topics:

- Overview of the HP-UX 11.0 release
- Benefits of 64-bit mode
- HP-UX languages available for 64-bit development
- Tools to help you port
- The HP-UX 32-bit and 64-bit cross-platform development environment
- Compatibility with previous releases

Overview of HP-UX 11.0

The HP-UX 11.0 release includes a 64-bit version and a 32-bit version of the operating system. This release of the operating system:

- Provides 64-bit addressing when the 64-bit version of HP-UX is installed on HP 64-bit hardware platforms. This enables programs to take advantage of very large address spaces and larger physical memory limits.
- Provides kernel level threads for maximum efficiency of multi-threaded applications.
- Complies with the latest NIS+ (Network Information Service) and NFS PV3 de facto standards for more secure network name services and larger network file systems.
- Runs in 32-bit mode on HP-UX 32-bit hardware platforms and in either 32-bit or 64-bit mode on HP-UX 64-bit hardware platforms.
- Supports run-time execution of both 32-bit and 64-bit applications on HP-UX 64-bit platforms.
- Supports inter-process communication between 32-bit and 64-bit applications via message queues, pipes, shared memory, and networking protocols.
- Provides a cross-platform development environment for developing 32-bit and 64-bit applications.

The following table shows the evolution of recent HP-UX releases:

Table 1-1 Capacity Limits of Recent HP-UX Releases

Attribute	10.01	10.10	10.20	64-bit 11.0
File System	4 GB	128 GB	128 GB	128 GB
File Size	2 GB	2 GB	128 GB local, 2 GB network	128 GB local and network
Physical RAM	2 GB	3.75 GB	3.75 GB	4 TB

Attribute	10.01	10.10	10.20	64-bit 11.0
Shared Memory	1.75 GB	1.75 GB	2.75 GB	8 TB
Process Data Space	.9 GB	1.9 GB	1.9 GB	4 TB
# File Descriptors	2 K	60 K	60 K	60 K
# of User Ids	60 K	60 K	~2,000 K	~2,000 K

Benefits for 64-bit Applications

Applications that are limited by the 32-bit address space need to transition to 64-bit mode. Potential examples include:

- database management systems
- engineering and mechanical design automation systems
- simulation and modeling programs
- decision support applications

The advantages of transitioning applications from 32-bit mode to 64-bit mode include:

- Large file caches on systems with large physical memory

  Database servers have improved performance when they can load significant portions of the database into memory.

- Large process data space mapped in a large virtual address space

  Simulation programs need to be able to map the entire simulation model into virtual memory.

- Large file support using standard system library calls

Benefits for 64-bit Applications

Some databases require data sets larger than 2 GB. It is simpler to store information for a large data set in a single file. 64-bit applications can use standard I/O routines to access files larger than 2 GB.

See Also:

See “HP-UX 64-bit Performance Considerations” on page 55 to learn about what impacts performance in 64-bit mode.

HP-UX Compilers

The following HP compiler products support both 64-bit and 32-bit program development:

- HP C
- HP aC++
- HP Fortran 90
- HP Assembler
- HP DDE (debugger bundled with compilers)
- HP PAK (performance toolkit bundled with compilers)
- HP Linker toolset (bundled with the operating system)

The following HP compiler products do not support 64-bit program development, but are available for 32-bit program development:

- HP Fortran 77
- HP Pascal
- HP MicroFocus COBOL
- HP C++ (cfront)

Transition Tools

There are several tools that can help you transition from previous releases of HP-UX:

- HP-UX Software Transition Toolkit (STK)

  Aids in transitioning your software to either the 32-bit or the 64-bit version of HP-UX 11.0. You can use its tools and documentation to identify and fix obsolete or changed APIs in C and C++ source code, scripts, and makefiles. To use the HP-UX STK, you must install it. The HP-UX STK is available free of charge on the HP-UX 11.0 Application Release CD-ROM, or via the World-Wide Web at http://www.software.hp.com/STK/.

- HP C

  Both `lint` and the HP C compiler provide options to help you transition your software to the HP-UX 64-bit data model. See Chapter 4, “Transitioning C and aC++ Programs to 64-bit Mode,” to learn how to identify and fix 64-bit porting issues.

- HP-UX System Release Notes

  Documents new, changed, and obsolete features, including:

  - system header file changes
  - system library changes
  - 64-bit versions of system libraries
  - source, data, and binary compatibility

- FlexeLint

  Identifies non-portable constructs in C and aC++ programs. FlexeLint is available from Gimpel Software.

Cross-Platform Development

HP-UX 11.0 provides a cross-platform development environment. You can compile and link both 32-bit and 64-bit applications on HP-UX 11.0 32-bit systems. Also, you can compile and link both 32-bit and 64-bit applications on HP-UX 11.0 64-bit systems.

You can optionally install 64-bit versions of HP-UX 11.0 system libraries on HP-UX 11.0 32-bit systems for cross-development. The 64-bit system libraries are in separate directories from the 32-bit system libraries.

Linking 32-bit and 64-bit object files (`.o`, `.sl`, `.a`) together is not supported. All modules in a program must be linked from either 32-bit objects or 64-bit objects.

You will need access to a 64-bit hardware platform running the 64-bit version of HP-UX 11.0 in order to test 64-bit programs.

Compiler Options for Cross Development

Unless specified, HP compilers generate object files that are compatible with the hardware on which you are compiling.

HP-UX 64-bit platforms use the PA-RISC 2.0 system architecture. The default compilation mode for these systems is 32-bit narrow mode for PA-RISC 2.0. PA-RISC 2.0 narrow mode programs only run on PA-RISC 2.0 systems. The compiler command line option for this mode is `+DA2.0` or `+DA2.0N`. (`+DA` means *destination architecture*.)

HP compilers generate 64-bit mode code when you specify the `+DA2.0W` command-line option. This is known as PA-RISC 2.0 wide mode. PA-RISC 2.0 wide mode programs only run on PA-RISC 2.0 systems running the 64-bit version of HP-UX.

Table 1-2 shows various compiler options for compiling to 32-bit and 64-bit mode and what systems the executables run on:

Table 1-2 Compiler Option / Hardware Architecture Run-time Compatibility

Option	PA-RISC 1.1 32-bit Platforms	PA-RISC 2.0 64-bit Platforms
`+DA1.1`	X	X
`+DAportable`	X	X
`+DA2.0` or `+DA2.0N`	N/A	X
`+DD64` [1] or `+DA2.0W` [2]	N/A	X

1. HP C supports this option for compiling in 64-bit mode.
2. HP aC++, HP C, and HP Fortran 90 support this option for compiling in 64-bit mode. (It is the same as +DD64.)

Please note the following cross-platform limitations:

- Debugging and program dump analysis tools for 64-bit programs are only supported on 64-bit HP-UX.
- Kernel cross-platform development is not supported. If the device driver refers to kernel header files and libraries, development must be done on the same platform as the target run-time platform.

Compiling in Networked Environments

When compiles are performed using diskless workstations or NFS-mounted file systems, the default code generation and scheduling is based on the local host processor. The system model numbers of the hosts where the source files reside do not affect the default code generation.

For More Information:

- See the `/opt/langtools/lib/sched.models` file for system model numbers, architectures, and processor names.
- Use the command `model` to determine the model number of your system.

Compatibility with Previous Releases

HP-UX 11.0 is binary compatible with HP-UX 10.*x*. Fully bound shared or archive applications that work on any HP-UX 10.*x* release continue to work on this release without recompiling, relinking, or modifying the application. A fully shared bound application consists of an executable program and all of its related user shared libraries.

See also the HP C, HP aC++, and HP Fortran 90 release notes for additional compatibility information.

2 Summary of Changes

Several changes and improvements have been made in support of the HP-UX 64-bit architecture. These changes are included in:

- HP C
- HP aC++
- HP Fortran 90
- System libraries
- Programming toolset
- Run time differences
- Application interoperability

HP C

To generate 32-bit mode code to run on HP-UX 64-bit systems, no new compiler command line options are required.

To compile in 64-bit mode, use the `+DD64` or `+DA2.0W` command line options.

NOTE

If you are porting from a previous release of HP-UX, be aware that extended ANSI mode (`-Ae`) is the default compilation mode since the HP-UX 10.30 release. See the *HP C Programmer's Guide* for information on how to port to ANSI mode.

The HP C compiler on HP-UX 11.0 includes support for both the 32-bit data model and the 64-bit data model. In 32-bit mode, `int`s, `long`s, and pointer types are 32 bits in size. In 64-bit mode, `long`s and pointers are 64-bits, and `int`s remain 32-bits. Table 2-1 shows the differences in C data type sizes and alignments:

Table 2-1 Differences between HP-UX 32-bit and 64-bit Data Models

Data Type	32-bit Mode Size (bits)	32-bit Mode Alignment (bits)	64-bit Mode Size (bits)	64-bit Mode Alignment (bits)
`int`	32	32	32	32
`long`	32	32	64	64
pointer	32	32	64	64

You may need to make source code changes, when transitioning to the HP-UX 64-bit data model, to correct assumptions made about the size and relationship of `int`, `long`, and pointer data types. Examples of programs that will require change include:

- Programs that assume that an `int` is the same size as a `long`.
- Programs that assume that an `int` is the same size as a pointer.
- Programs that perform arithmetic or comparison operations between `int`s, `long`s and pointers, and between signed numeric types and unsigned numeric types.

- Programs that make assumptions about data item sizes and alignment in structures.
- Programs that use hard-coded constants.

HP C 11.0 New Features

The following are new or changed HP C features included in the HP-UX 11.0 release:

Table 2-2 New and Changed HP C Features

Feature	What it Does
`+DD64`	Recommended option for compiling in 64-bit mode on the PA-RISC 2.0 architecture. The macros `__LP64__` and `_PA_RISC2_0` are `#defined`. (Same as `+DA2.0W`.)
`+DA2.0W`	Compiles in 64-bit mode for the PA-RISC 2.0 architecture. The macros `__LP64__` and `_PA_RISC2_0` are `#defined`. (Same as `+DD64`.)
`+DA2.0N`	Compiles in 32-bit mode (narrow mode) for the PA-RISC 2.0 architecture. The macro `_PA_RISC2_0` is `#defined`. (Same as `+DA2.0`.)
`+DD32`	Compiles in 32-bit mode and creates code compatible with PA-RISC 1.1 architectures. (Same as `+DA1.1` and `+DAportable`.)
`+hugesize`	Lowers the threshold for huge data objects allocated to the huge data space (`.hbss`).
`-dynamic`	Creates dynamically bound executables. The linker will look for shared libraries first and then archive libraries. This option is on by default when you compile in 64-bit mode.
`-noshared`	Creates statically bound executables. You cannot link to shared libraries if you specify this option.
`+M1`	Turns on platform migration warnings. These features may be unsupported in a future release.
`+M2`	Turns on HP-UX 64-bit data model warnings. (Use this option with the `+DA2.0W` or `+DD64` options.)

Feature	What it Does
`__LP64__`	Macro that is automatically defined by the HP C compiler when compiling in 64-bit mode. Can be used within conditional directives to isolate 64-bit mode code.
`PACK` or `HP_ALIGN` pragmas	Data alignment pragmas. The `HP_ALIGN` pragma includes support for 64-bit mode. The new `PACK` pragma provides a convenient way of specifying alignment.
`lint`	Identifies non-portable constructs. Use the `+DD64` and `+M2` options to `lint` when transitioning to the HP-UX 64-bit data model.

For More Information:

- See Chapter 3, "HP-UX 64-bit Porting Concepts."
- See Chapter 4, "Transitioning C and aC++ Programs to 64-bit Mode."
- See the *HP C Online Reference* (`cc +help`) for information about advanced 64-bit optimization options.
- See the *HP C/HP-UX Release Notes* for information about huge data.

HP aC++

To generate 32-bit mode code to run on HP-UX 64-bit systems, no new compiler command line options are required.

To compile in 64-bit mode, use the `+DA2.0W` command line option.

NOTE

Applications written in HP C++ (cfront) must be migrated to aC++ prior to compiling in 64-bit mode. For information on migrating to aC++, see *HP aC++ Migration Guide*, URL: http://www.hp.com/lang/cpp/.

The aC++ compiler on HP-UX 11.0 includes support for both the 32-bit data model and the 64-bit data model. In 32-bit mode, integer, long, and pointer types are 32 bits in size. In 64-bit mode, long and pointer types are 64 bits in size, and integers are 32 bits.

The following new HP aC++ features are included in the 11.0 release:

Table 2-3 New aC++ Features

Feature	What it Does
`+DA2.0W`	Compiles in 64-bit mode for the PA-RISC 2.0 architecture. The macros `__LP64__` and `_PA_RISC2_0` are `#defined`.
`+DA2.0N`	Compiles in 32-bit mode for the PA-RISC 2.0 architecture. The macro `_PA_RISC2_0` is `#defined`. (Same as `+DA2.0`.)
`+hugesize`	Lowers the threshold for huge data objects allocated to the huge data space (`.hbss`).
`__LP64__`	Macro that is automatically defined by the HP aC++ compiler when compiling in 64-bit mode. Can be used within conditional directives to isolate 64-lbit mode code.

For More Information:

- See Chapter 3, “HP-UX 64-bit Porting Concepts.”
- See Chapter 4, “Transitioning C and aC++ Programs to 64-bit Mode.”
- See the *HP aC++ Release Notes* for information about huge data.

HP Fortran 90

To generate 32-bit mode code to run on HP-UX 64-bit systems, no new compiler command line options are required.

To compile in 64-bit mode, use the `+DA2.0W` command line option.

There are no HP Fortran 90 language differences between 32-bit and 64-bit programs. Recompiling should suffice to convert a 32-bit Fortran program to run as a 64-bit program.

HP Fortran 90 and HP C Data Types

Whereas using the `+DA2.0W` option to compile HP Fortran 90 programs in 64-bit mode has no effect on Fortran data types, the C language has some differences in data type sizes. If your Fortran program calls functions written in C and is compiled in 64-bit mode, the size differences may require promoting data items that are passed to or from the C functions.

Table 2-4 shows the differences between the corresponding data types in HP Fortran 90 and C when compiling in 32-bit mode and in 64-bit mode. Table 2-5 on page 25 shows the differences when the Fortran program is compiled with the `+autodbl` option. (The `+autodbl` option increases the default size of integer, logical, and real items to 8 bytes, and double precision and complex items for 16 bytes.)

Table 2-4 Size Differences Between HP Fortran 90 and C Data Types

HP Fortran 90 Data Types	C Data Types		Sizes (in Bits)
	32-Bit Mode	64-Bit Mode	
`INTEGER`	`int` or `long`	`int`	32
`INTEGER*4`	`int` or `long`	`int`	32
`INTEGER*8`	`long long`	`long` or `long long`	64

HP Fortran 90 Data Types	C Data Types		Sizes (in Bits)
	32-Bit Mode	64-Bit Mode	
REAL	float	float	32
DOUBLE PRECISION	double	double	64
REAL*16	long double	long double	128

Table 2-5 Size Differences After Compiling with +autodbl

HP Fortran 90 Data Types	C Data Types		Sizes (in Bits)
	32-Bit Mode	64-Bit Mode	
INTEGER	long long	long	64
INTEGER*4	int or long	int	32
INTEGER*8	long long	long	64
REAL	double	double	64
DOUBLE PRECISION	long double	long double	128
REAL*16	long double	long double	128

New Fortran 90 Features

The following are new features included in the HP-UX 11.0 release:

Table 2-6 New and Changed HP Fortran 90 Features

Feature	What it Does
+DA2.0W	Compiles in 64-bit mode for the PA-RISC 2.0 architecture.

Feature	What it Does
`+DA2.0N`	Compiles in 32-bit mode (narrow mode) for the PA-RISC 2.0 architecture.
`+hugesize`	Lowers the threshold for huge `COMMON` blocks allocated to the huge data space (`.hbss`).
`+hugecommon =`*name*	Allocated specific `COMMON` blocks to the huge data space (`.hbss`).

In addition, HP Fortran 90 adds new parallelization directives, library calls, fast math intrinsics, and optimization options.

For More Information:

- See the *HP Fortran 90 Release Notes* for information about huge data.
- See the *HP Fortran 90 Programmer's Reference* for information about command-line options.

Programming Toolset

Table lists HP-UX programming tools and shows whether they support 32-bit and 64-bit programs:

HP-UX Programming Tools

Tool	What it Does	32-bit Support	64-bit Support
`ar`	Creates an archive library.	Yes	Yes
`chatr`	Changes an executable file's internal attributes.	Yes	Yes
`elfdump`	Displays information about a 64-bit ELF object file.	No	Yes
`fastbind`	Improves start-up time of programs that use shared libraries.	Yes	Yes
`file`	Determines a file type and lists its attributes.	Yes	Yes
`getconf`	Gets configurable system information.	Yes	Yes
HP DDE debugger [1]	Helps you find run-time errors in programs.	Yes	Yes
HP GDB debugger (vers.1.0) [1]	Helps you find run-time errors in programs.	Yes	No
HP PAK: `puma`, `ttv`[1]	Analyzes program performance. Puma displays program performance based on statistical samplings. TTV displays thread traces.	Yes	Yes
CXperf[2]	Creates a profile of program performance statistics.	Yes	Yes

Tool	What it Does	32-bit Support	64-bit Support
`lint` [3]	Detects defects, non-portable, and inefficient code in C programs.	Yes	Yes
`ldd`	Shows shared libraries used by a program or shared library.	No	Yes
`make`	Manages program builds.	Yes	Yes
`nm`	Displays symbol table information.	Yes	Yes
profilers: `prof`, `gprof`	Helps you locate parts of a program most frequently executed. Using this data, you may restructure programs to improve performance.	Yes	Yes
`size`	Prints text, data, and bss (uninitialized data) section sizes of an object file.	Yes	Yes
`strip`	Strips symbol table and line numbers from an object file.	Yes	Yes

1. Bundled with compilers. Tools that are not footnoted are bundled with the OS.
2. CXperf is only supported on HP 9000 K-class and V-class servers. Available separately. Contact your HP sales office.
3. Included in the HP C/ANSI C Developer's Bundle.

64-bit Linker Toolset Features

The linker toolset provides the following new features for developing 64-bit programs:

Table 2-7 Summary of New Linker 64-bit Toolset Features

64-bit Feature	Description
dlopen(3X) family of dynamic loading routines [1]	Routines for manipulating shared libraries.
libelf() library of routines	Routines for manipulating the 64-bit ELF object file format. Includes the *nlist64()* routine to dump symbol information.
`elfdump`	A tool that displays information about a 64-bit ELF object file.
`ldd`	A tool that shows shared libraries used by a program or shared library.
New options to `ld` and `chatr`	Command line options to assist in the development of 64-bit applications.
Standard SVR4 dynamic loading features	Includes SVR4 dynamic path searching and breadth-first symbol searching.
Mapfile support	A linker option that lets you control the organization of segments in executable files. This feature is intended for embedded systems development.

1. SVR4 compatible feature.

For More Information:

See the *Linker and Libraries Online User Guide* (`ld +help`).

Link Time Differences

Table 2-8 lists linker features that are not available in 64-bit mode:

Table 2-8 Unsupported Linker Features in 64-bit Mode

Option or Behavior	Description
`-A` *name*	Specifies incremental loading. 64-bit applications must use shared libraries instead.
`-C` *n*	Does parameter type checking. This option is unsupported.
`-S`	Generates an initial program loader header file. This option is unsupported.
`-T`	Saves data and relocation information in temporary files to reduce virtual memory requirements during linking. This option is unsupported.
`-q`, `-Q`, `-n`	Generates an executable with file type `DEMAND_MAGIC`, `EXEC_MAGIC`, and `SHARE_MAGIC` respectively. These options have no effect and are ignored in 64-bit mode.
`-N`	Causes the data segment to be placed immediately after the text segment. This option is accepted but ignored in 64-bit mode. If this option is used because your application data segment is large, then the option is no longer needed in 64-bit mode. If this option is used because your program is used in an embedded system or other specialized application, consider using mapfile support with the `-k` option.
`+cg` *pathname*	Specifies *pathname* for compiling I-SOMs to SOMs. This option is unsupported.

Option or Behavior	Description
+dpv	Displays verbose messages regarding procedures which have been removed due to dead procedure elimination. Use the -v linker option instead.
intra-library versioning	Specified by using the HP_SHLIB_VERSION pragma (C and aC++) or SHLIB_VERSION directive (Fortran90). In 32-bit mode, the linker lets you version your library by object files. 64-bit applications must use SVR4 library-level versioning instead.
Duplicate code and data symbols	Code and data cannot share the same namespace in 64-bit mode. You should rename the conflicting symbols.
All internal and undocumented linker options	These options are unsupported.

For More Information:

See the *Linker and Libraries Online User Guide* (ld +help).

Run Time Differences

Applications compiled and linked in 64-bit mode use a run-time dynamic loading model similar to other SVR4 systems. There are two main areas where program start-up changes in 64-bit mode:

- Dynamic path searching for shared libraries
- Symbol searching in dependent libraries

It is recommended that you use the standard SVR4 linking option (`+std`, which is on by default) when linking 64-bit applications. If there are circumstances during the transition when you need 32-bit compatible linking behavior, use the `+compat` option. This option forces the linker to use 32-bit linking and dynamic loading behavior.

The following table summarizes the dynamic loader differences between 32-bit and 64-bit mode:

Table 2-9 Dynamic Loading Differences

Linker and Loader Functions	32-bit Mode Behavior	64-bit Mode Behavior
`+s` and `+b` *path_list* ordering	Ordering is significant.	Ordering is insignificant by default. Use `+compat` to enforce ordering.
Symbol searching in dependent libraries	Depth first search order.	Breadth first search order. Use `+compat` to enforce depth first ordering.
Run time path environment variables	No run time environment variables are available by default. If `+s` is specified, then `SHLIB_PATH` is available.	`LD_LIBRARY_PATH` and `SHLIB_PATH` are available. Use `+noenv` or `+compat` to turn off run-time path environment variables.

Linker and Loader Functions	32-bit Mode Behavior	64-bit Mode Behavior
+b *path_list* and -L *directories* interaction	-L directories recorded as absolute paths in executables.	-L directories are not recorded in executables. Add all directories specified in -L to +b *path_list*.

For More Information:

- See "Dynamic Path Searching for Shared Libraries" on page 33.
- See "Symbol Searching in Dependent Libraries" on page 35.

Dynamic Path Searching for Shared Libraries

Dynamic path searching is the process that allows the location of shared libraries to be specified at run time. In 32-bit mode, you can enable run-time dynamic path searching of shared libraries in two ways:

- by linking the program with +s, enabling the program to use the path list defined by the SHLIB_PATH environment variable at run time.
- by storing a directory path list in the program with the linker option +b *path_list*.

If +s or +b *path_list* is enabled, all shared libraries specified with the -l*library* or -l:*library* linker options are subject to a dynamic path lookup at run time.

In 64-bit mode, the dynamic path searching behavior has changed:

- The +s dynamic path searching option is enabled by default. It is not enabled by default in 32-bit mode.
- The LD_LIBRARY_PATH environment variable is available in addition to the SHLIB_PATH environment variable.
- An embedded run-time path list called RPATH may be stored in the executable. If +b *path_list* is specified at link time, these directories are added to RPATH. If +b *path_list* is not specified, the linker creates a default RPATH consisting of:

  1. directories in the -L option (if specified), followed by
  2. directories in the LPATH environment variable (if specified).

- By default, in 64-bit mode, the linker ignores the ordering of the +b *path_list* and +s options.
- At run time, the dynamic loader searches directory paths in the following order:
  - LD_LIBRARY_PATH (if set), followed by
  - SHLIB_PATH (if set), followed by
  - RPATH, followed by
  - the default locations /lib/pa20_64 and /usr/lib/pa20_64.

Examples

The following are examples of specifying library paths in 32-bit and 64-bit mode:

- Linking to libraries by fully qualifying paths:

  In this example, the program is linked with /opt/myapp/mylib.sl:

  ```
  $ cc main.o /opt/myapp/mylib.sl          Perform 32-bit link.
  $ cc +DD64 main.o /opt/myapp/mylib.sl    Perform 64-bit link.
  ```

  At run-time, in both 32-bit and 64-bit mode, the dynamic loader only looks in /opt/myapp to find mylib.sl.

- Linking to libraries using the -l*library* or -l:*library* options:

  In this example, the +s option is not explicitly enabled at link time. Two versions of a shared library called libfoo.sl exist; a 32-bit version in /usr/lib and a 64-bit version in /usr/lib/pa20_64:

  ```
  $ cc main.o -lfoo -o main          Perform 32-bit link.
  ```

  When linked in 32-bit mode, main will abort at run time if libfoo.sl is moved from /usr/lib. This is because the absolute path name of the shared library /usr/lib/libfoo.sl is stored in the executable.

  ```
  $ cc +DD64 main.o -lfoo -o main    Perform 64-bit link.
  ```

  When linked in 64-bit mode, main will not abort at run time if libfoo.sl is moved, as long as SHLIB_PATH or LD_LIBRARY_PATH is set and point to libfoo.sl.

- Linking to libraries using -L and +b *path_list*:

The `-L` option is used by the linker to locate libraries at link time. The `+b` option is used to embed a library path list in the executable for use at run time.

Here is the 32-bit mode example:

```
$ cc main.o -L. -Wl,+b/var/tmp -lme     Link the program.
$ mv libme.sl /var/tmp/libme.sl         Move libme.sl.
$ a.out                                 Run the program.
```

In 32-bit mode, the dynamic loader searches paths to resolve external references in the following order:

1. `/var/tmp` to find `libme.sl` *found*
2. `/var/tmp` to find `libc.sl` *not found*
3. `/usr/lib/libc.sl` *found*

Here is the 64-bit mode example:

```
$ cc +DD64 main.o -L. -Wl,+b/var/tmp -lme   Link the program.
$ mv libme.sl /var/tmp/libme.sl             Move libme.sl.
$ a.out                                     Run the program.
```

In 64-bit mode, the dynamic loader searches paths to resolve external references in the following order:

1. `LD_LIBRARY_PATH` (if set) to find `libme.sl` *not found*
2. `SHLIB_PATH` (if set) to find `libme.sl` *not found*
3. `/var/tmp` to find `libme.sl` *found*
4. `LD_LIBRARY_PATH` (if set) to find `libc.sl` *not found*
5. `SHLIB_PATH` (if set) to find `libc.sl` *not found*
6. `/var/tmp` to find `libc.sl` *not found*
7. `/usr/lib/pa20_64/libc.sl` *found*

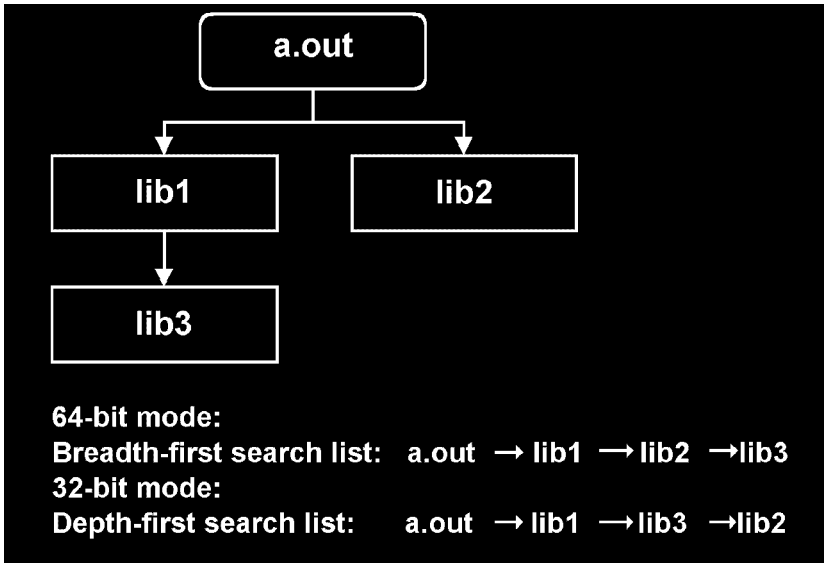
Symbol Searching in Dependent Libraries

In 64-bit mode, the dynamic loader searches shared libraries using a **breadth-first** search order. Breadth-first symbol searching is used on all SVR4 platforms.

In 32-bit mode, the dynamic loader searches shared libraries using a **depth-first** search order.

Figure 2-1 shows an example program with shared libraries and compares the two search methods:

Figure 2-1 Search Order of Dependent Libraries



The commands to build the libraries and the executable in Figure 2-1 are shown:

```
ld -b lib2.o -o lib2.sl
ld -b lib3.o -o lib3.sl
ld -b lib1.o -L. -l3 -o lib1.sl
cc main.o -Wl,-L. -l1 -l2 -o main
```

In 32-bit mode, if a procedure called `same_name()` is defined in both `lib3.sl` and `lib2.sl`, `main` calls the procedure defined in `lib3.sl`. In 64-bit mode, `main` calls the procedure in `lib2.sl`.

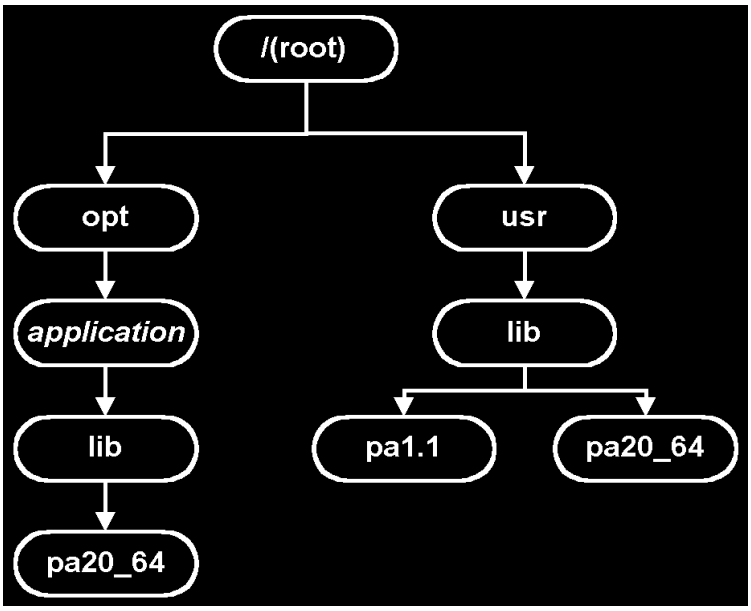
System Libraries

HP-UX 64-bit systems provide a new subdirectory called `pa20_64` for 64-bit versions of system and HP product libraries.

The 64-bit file system layout leaves the current 32-bit directory structure intact. This helps preserve binary compatibility with 32-bit versions of shared libraries whose paths are embedded in executables.

Figure 2-2 shows the new directory structure:

Figure 2-2 New Subdirectory for 64-bit Libraries (pa20_64)



The linker automatically finds the correct set of system libraries depending on whether the application is compiled in 32-bit or 64-bit mode.

Library providers are encouraged to supply both 32-bit and 64-bit versions of application libraries. Be sure to develop a strategy for library naming conventions, directory structures, link-time options, and run-time environment variables.

32-bit and 64-bit Application Interoperability

Some restrictions apply when sharing objects, such as data and memory, between 32-bit applications and 64-bit applications. These restrictions also apply when sharing objects between 32-bit applications and the 64-bit version of the operating system.

This section summarizes topics described in the *HP-UX Application Interoperability White Paper*, URL:http://www.software.hp.com/STK/.

General Issue

You should ensure that all data shared between a 64-bit and 32-bit application is of the same size and alignment within both applications.

Shared Memory

32-bit applications can only attach to shared memory segments which exist in a 32-bit virtual address space. 64-bit applications can attach to shared memory segments in a 32-bit or 64-bit virtual address space. To create a shareable memory segment between 32-bit and 64-bit applications, your 64-bit application must specify the `IPC_SHARE32` flag with the `IPC_CREAT` flag when invoking `shmget(2)`. The `IPC_SHARE32` flag causes the shared memory segment to be created in a 32-bit address space.

Message Queues

The size of a message queue is defined as type `size_t`. If your 64-bit application exchanges data with 32-bit applications via message queues, make sure that the size of the message does not exceed the largest 32-bit unsigned value.

Memory-Mapped Files

32-bit applications can only share memory-mapped files that are mapped into a 32-bit virtual address space. 64-bit applications can share memory-mapped files in a 32-bit or 64-bit virtual address space. To map

a file into memory that is shareable between 64-bit and 32-bit applications, your 64-bit application must specify the `MAP_ADDR32` flag with the `MAP_SHARED` flag when invoking `mmap(2)`.

nlist

Symbols within 64-bit executables on 64-bit HP-UX are assigned 64-bit values. An application extracting 64-bit values from the symbol table of a 64-bit executable needs 64-bit data fields. 32-bit mode applications must either be ported to 64-bit mode in order to extract 64-bit symbols, or must use the `nlist64`(3C) function to accomplish this task.

X11/graphics

Although the X-server is supported on both the 32-bit and 64-bit versions of HP-UX 11.0, the graphical user interface (GUI) for the client application must be a 32-bit application.

64-bit versions of graphical libraries, such as X11/Motif and all 3D libraries are not provided on HP-UX 11.0 since large memory and process data space are not needed in the GUI component of the application.

Many large applications already support the GUI component for the application in a separate process from the backend processing component and communicate via standard interprocess communication mechanisms.

If you are converting your application to a 64-bit application, and the GUI and backend are in separate processes, follow these guidelines:

- Leave the GUI component as a 32-bit application.
- Convert the backend process, which may need to take advantage of more than 4GB of memory or process data space, to a 64-bit process.

If the GUI component and the processing component are in the same process, the GUI component can be split into a separate process and can communicate with the back-end processing component via standard interprocess communication mechanisms.

Large Files

32-bit applications can open, create and use large files. A large file is a file that is 2GBs or greater. However, when creating or opening large files, your 32-bit application must specify the `O_LARGEFILE` flag with the `open(2)` system call.

Also, using `lseek(2)` within a 32-bit application to position a file pointer beyond 2GB produces undefined results. You should use the `lseek64(2)` interface instead.

For detailed information, see the HP-UX 11.0 white paper titled *HP-UX Large Files* in `/usr/share/doc/lg_files.ps`.

pstat

The following *pstat_get**(2) system calls may fail, with `errno` set to `EOVERFLOW`, when invoked within 32-bit applications. This is because within 64-bit HP-UX, many parameters, limits and addresses are 64-bit values and they cannot fit into fields of the corresponding `struct pst_*` data structure.

```
pstat_getdynamic(2)
pstat_getipc(2)
pstat_getproc(2)
pstat_getprocvm(2)
pstat_getshm(2)
pstat_getfile(2)
```

3 HP-UX 64-bit Porting Concepts

This chapter describes porting concerns of the HP-UX 64-bit data model and performance considerations when transitioning to 64-bit platforms.

The following topics are included:

- The ILP32 and LP64 data models
- ILP32 to LP64 porting concerns
- Architecture-specific changes
- HP-UX 64-bit performance considerations

ILP32 and LP64 Data Models

The ANSI/ISO C standard specifies that C must support four signed and four unsigned integer data types: `char`, `short`, `int`, and `long`. There are few requirements imposed by the ANSI standard on the sizes of these data types. According to the standard, `int` and `short` should be at least 16 bits; and `long` should be at least as long as `int`, but not smaller than 32 bits.

Traditionally, Kernighan and Ritchie (K&R) C assumes `int` is the most efficient or *fastest* integer data type on a machine. ANSI C, with its integral promotion rule, continues this assumption.

The HP-UX 32-bit data model is called **ILP32** because `int`s, `long`s, and pointers are 32 bits.

The HP-UX 64-bit data model is called **LP64** because `long`s and pointers are 64 bits. In this model, `int`s remain 32 bits.

NOTE The LP64 data model is the emerging standard on 64-bit UNIX systems provided by leading system vendors. Applications that transition to the LP64 data model on HP-UX systems are highly portable to other LP64 vendor platforms.

Data Type Sizes

The size of the base HP C data types under the HP-UX implementation of ILP32 and LP64 are shown in Table 3-1:

Table 3-1 HP C/HP-UX 32-bit and 64-bit Base Data Types

Data Type	ILP32 Size (bits)	LP64 Size (bits)
`char`	8	8
`short`	16	16
`int`	32	32
`long`	32	64
`long long` [1]	64	64

Data Type	ILP32 Size (bits)	LP64 Size (bits)
pointer	32	64
`float`	32	32
`double`	64	64
`long double`	128	128
`enum` [2]	32	32

1. The `long long` data type is an HP value-added extension.
2. Sized `enum`s are available in 32-bit and 64-bit mode.

Huge Data

In general, huge data is any data that is larger than can be represented on a 32-bit system. Hence, huge data is only supported on 64-bit systems.

More specifically, huge data is any data greater than a certain size placed into a huge data segment (hbss segment). Smaller objects are placed into a bss segment.

In general, data objects on 32-bit systems can be as large as 2^{28} bytes or 256 megabytes whereas on 64-bit systems data objects can be as large as 2^{58} bytes or larger in some cases.

HP C/HP-UX supports uninitialized arrays, structs, and unions to a maximum of 2^{58} bytes. HP aC++ supports uninitialized arrays and C-style structs and unions to a maximum of 2^{61} bytes.

For More Information For details see the *HP C/HP-UX Release Notes*, the *HP aC++ Release Notes*, or the *HP Fortran 90 Release Notes*.

ILP32 to LP64 Porting Concerns

Some fundamental changes occur when moving from the ILP32 data model to the LP64 data model:

- `long`s and `int`s are no longer the same size.
- pointers and `int`s are no longer the same size.
- pointers and `long`s are 64 bits and are 64-bit aligned.
- Predefined types `size_t` and `ptrdiff_t` are 64-bit integral types.

These differences can potentially impact porting in the following areas:

- Data truncation
- Pointers
- Data type promotion
- Data alignment and data sharing
- Constants
- Bit shifts and bit masks
- Bit fields
- Enumerated types

See Also:

See Chapter 4, “Transitioning C and aC++ Programs to 64-bit Mode,” to learn how to identify and fix 64-bit porting issues.

Data Truncation

Truncation problems can happen when assignments are made between 64-bit and 32-bit data items. Since `int`s, `long`s, and pointers are 32 bits in ILP32, mixed assignments between these data types do not present any special concerns. However, in the LP64 data model, `long`s and pointers are no longer the same size as `int`s. In LP64, truncation will occur when pointers or `long`s are assigned to `int`s.

In LP64, truncation can occur during:

- initialization
- assignments
- parameter passing
- return statements

- casts

Pointers and `long`s are not the only data types whose size has changed. Some data types defined in header files that are 32 bits under ILP32 — for example, `off_t` — are now 64 bits. Variables declared with `off_t` may be truncated when assigned to `int`s in LP64.

Pointers

Avoiding pointer corruption is an important concern when migrating to LP64:

- Assigning a 32-bit hexadecimal constant or an `int` to a pointer type will result in an invalid address and may cause errors when the pointer is dereferenced.
- Casting a pointer to an `int` results in truncation.
- Casting an `int` to a pointer may cause errors when the pointer is dereferenced.
- Functions that return pointers, when declared improperly, may return truncated values.
- Comparing an `int` to a pointer may cause unexpected results.

Pointer arithmetic is a source of difficulty in migration.

Standard C behavior increments a pointer by the size of the data type to which it points. This means if the variable `p` is a pointer to `long`, then the operation `(p + 1)` increments the value of `p` by 4 bytes in ILP32 and by 8 bytes in LP64.

Casts between `long*` to `int*` are problematic because the object of a long pointer is 64 bits in size, but the object of an int pointer is only 32 bits in size.

Data Type Promotion

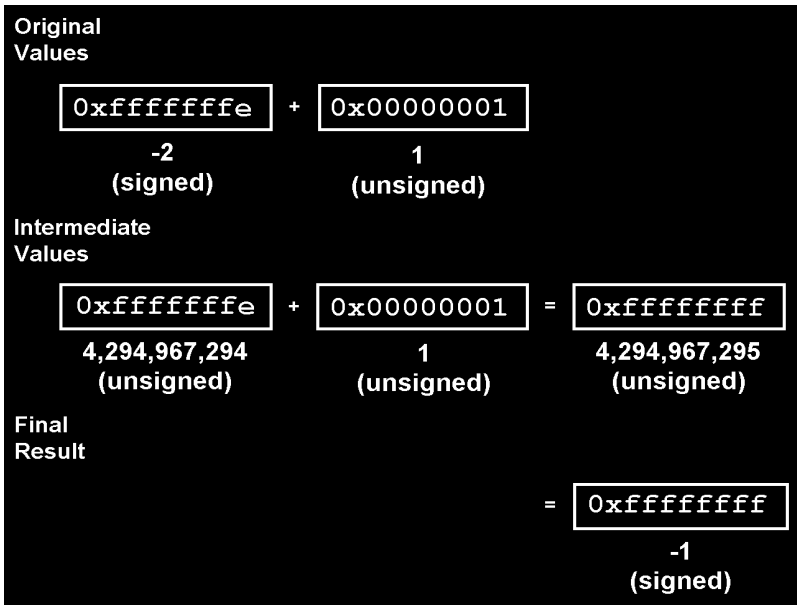
When comparisons and arithmetic operations are performed between variables and constants with different data types, ANSI C first converts these types to compatible types. For example, when a `short` is compared to a `long`, the `short` is first converted to a `long`. This conversion process is called **data type promotion**.

Certain data type promotions result in signed numbers being treated as unsigned numbers. When this happens, you can occasionally get unexpected results. For example:

```
long result;
int i = -2;
unsigned int j = 1;
result = i + j;
```

In ANSI C under the 32-bit data model, the results are shown:

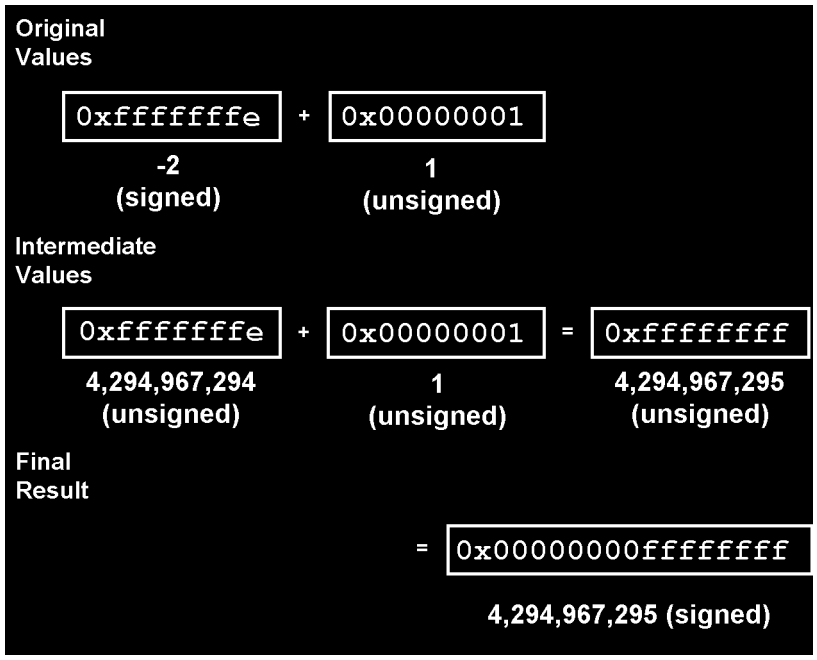
Figure 3-1 Data Type Promotion Example in ILP32



The intermediate result (an `unsigned int`) and the final result (a signed `long`) have the same internal representation because they are both 32 bits. Since the final result is signed, the answer is `-1`.

In ANSI C under the 64-bit data model, the results are different:

Figure 3-2 Data Type Promotion Example in LP64



When the 32-bit intermediate result (an `unsigned int`) is converted to the 64-bit final result (a `signed long`), the left 32 bits are zero-filled. This results in a very large 64-bit positive number.

Data Alignment and Data Sharing

Data alignment rules determine where fields are located in memory. There are differences between the LP64 data alignment rules and the ILP32 data alignment rules.

In ILP32, pointers and `long`s are 32 bits and are aligned on 32-bit boundaries. In LP64, pointers and `long`s are 64 bits and are aligned on 64-bit boundaries.

Applications that do not consider alignment differences between ILP32 and LP64 can have trouble sharing binary data. Data exchanged between ILP32 and LP64 mode programs, whether via files, remote procedure calls, or other messaging protocols, may not be aligned as expected.

Table 3-2 shows the data alignment for C data types:

Table 3-2 ILP32 and LP64 Data Alignment

Data Type	ILP32 Size (bytes)	ILP32 Alignment	LP64 Size (bytes)	LP64 Alignment
`char`	1	1-byte	1	1-byte
`short`	2	2-byte	2	2-byte
`int`	4	4-byte	4	4-byte
`long`	4	4-byte	8	8-byte
`long long`	8	8-byte	8	8-byte
`pointer`	4	4-byte	8	8-byte
`float`	4	4-byte	4	4-byte
`double`	8	8-byte	8	8-byte
`long double`	16	8-byte	16	16-byte
`struct`	depends on members [1]	depends on members	depends on members	depends on members
`enum`	4	4-byte	4	4-byte

1. aligned on the same boundary as its most strictly aligned member.

Structure Member Alignment

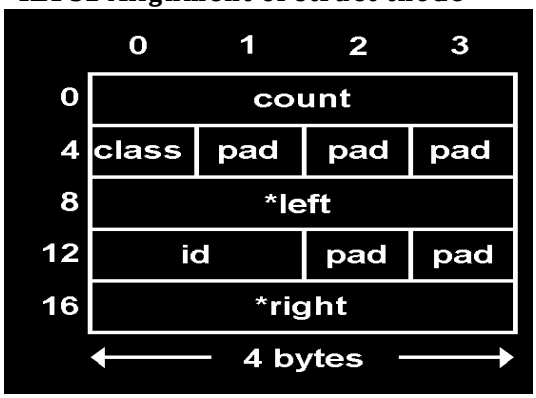
Data alignment of structures is affected by porting from ILP32 to LP64. Structure members may be padded differently in ILP32 and LP64 in order for the structure members to begin on specific alignment boundaries.

Here is an example structure that is aligned differently for ILP32 and LP64:

```
struct tnode {
   long count;
   char class;
   struct tnode *left;
   short id;
   struct tnode *right;
}
```

The tnode structure is aligned according to the alignment shown in Table 3-2. Figure 3-3 shows the alignment for tnode in ILP32:

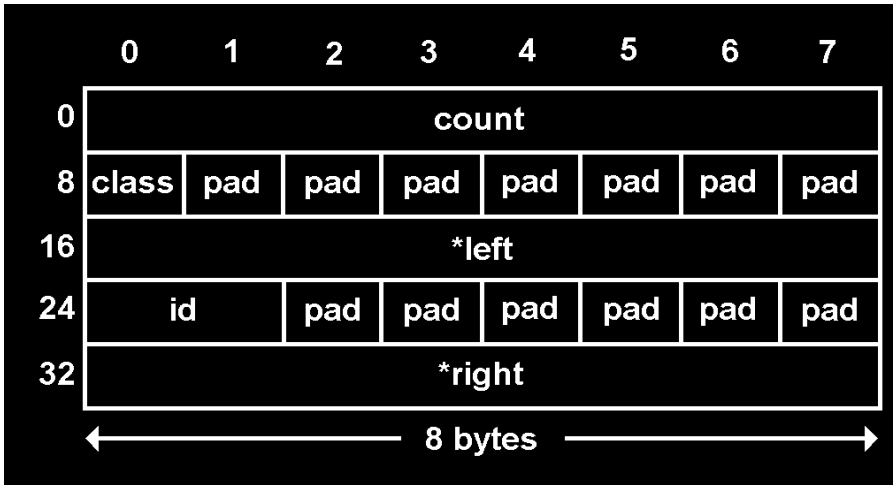
Figure 3-3 ILP32 Alignment of struct tnode



In ILP32, this data structure contains 20 bytes.

Figure 3-4 shows the alignment for the tnode structure in LP64.

Figure 3-4 LP64 Alignment of struct tnode



In LP64, this data structure contains 40 bytes.

In the example shown, the same structure definition has different sizes and the structure members have different offsets.

For More Information:

For information on how to create portable data structures, see the *HP C Programmer's Guide* or *HP-UX Applications Interoperability White Paper*, URL://www.software.hp.com/STK/.

Constants

When a program with hexadecimal constants is ported from ILP32 to LP64, the data types assigned to the constants may change. The following table illustrates some common hex constants and their types:

Constant	ANSI C ILP32	ANSI C LP64
`0x7fffffff`	`int`	`int`

Constant	ANSI C ILP32	ANSI C LP64
`0x7fffffffL`	`long`	`long`
`0x80000000`	`unsigned int`	`unsigned int`
`0x80000000L`	`unsigned long`	`long`

In LP64, 32-bit hexadecimal constants may no longer set pointers or masks to the correct value. In LP64, the first 32 bits of 64-bit pointers contain significant information.

Bit Fields

Unqualified bit fields are unsigned by default in LP64. In ILP32, unqualified bit fields are signed by default.

Bit fields of enumerated types are signed if the enumeration base type is signed and unsigned if the enumeration base type is unsigned.

Unnamed, non-zero length bit fields do not affect the alignment of a structure or union in LP64. In ILP32, unnamed, non-zero length bit fields affect the alignment of structures and unions.

Bit Shifts and Bit Masks

Bit shifts and bit masks are sometimes coded with the assumption that the operations are performed in variables that have the same data type as the result. In cases such as:

`a = b` *operation* `c`

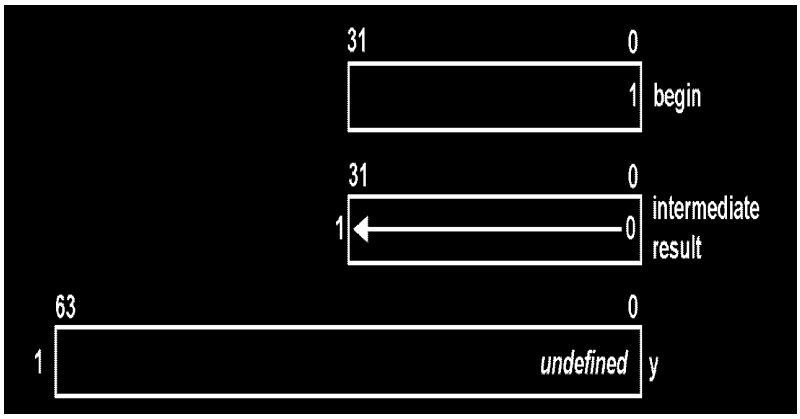
the data type used for the intermediate result of the operation depends on the types of `b` and `c`. The intermediate result is then promoted to the type of `a`. If the result requires 64 bits, but `b` and `c` are 32-bit data types, then the intermediate result either overflows or is truncated before being assigned to `a`.

In the following example, the left operand `1` is a small numeric constant which the compiler treats as a 32-bit value in both ILP32 and LP64:

```
unsigned long y;
y = (1 << 32);    /* Overflows in both data models. */
```

This bit shift uses a 32-bit data type as the intermediate result. In 64-bit mode, the operation overflows and the final result is *undefined* as shown:

Figure 3-5 Bit Shift Overflow in LP64



You can use suffixes such as `L` and `UL` for `long` and `unsigned long` if you need `long` constants. For example, in 64-bit mode, the above code fragment can be changed to:

```
y = (1L << 32);  /* 2^32 in LP64. Overflows in ILP32. */
```

Enumerated Types

In LP64, enumerated types are signed only if one or more of the enumeration constants defined for that type is negative. If all enumeration constants are non-negative, the type is unsigned. In ILP32, enumerated types are always signed.

Architecture Specific Changes

There is a class of porting issues that is not strictly caused by the 64-bit architecture, but is a side-effect of the 64-bit architecture.

Assembly Language

Assembly language code may need changes due to the 64-bit PA-RISC 2.0 calling conventions. You may also want to take advantage of the new instructions for improved performance.

The following summarizes items that may need adjustments:

- The procedure calling conventions are different. For example, the number of items passed on the stack may be different.
- Instead of `ldw` and `stw`, use `ldd` and `std` when loading and storing 64-bit values.
- Addresses are capable of holding 64-bit values.
- The 64-bit ELF object file format is more restrictive than the 32-bit object file format. Therefore, the set of legal instructions is more restrictive.
- Instead of `.word`, use the `.dword` pseudo-op when allocating storage for a pointer.
- Alignment of data items may be different.
- In 64-bit mode, the Assembler ignores the `.CALL` directive. This means the linker does not ensure that the caller and called procedure agree on argument locations. If you do not know the prototype of the called procedure, you must pass floating point parameters in both the corresponding general registers and corresponding floating-point registers.

For More Information:

- See the *HP Assembler Reference Manual* for more details about 64-bit Assembler changes.
- See *ELF-64 Object File Format*, URL:http://www.software.hp.com/STK/.

Object File Format

HP PA-RISC 1.0 and 1.1-based systems use the System Object Module (SOM) object file format. This is a proprietary format. It is the common representation of code and data for all compilers which generate code for PA-RISC 1.*x*-based systems.

HP PA-RISC 2.0-based systems use the SOM object file format in 32-bit mode and the industry-standard Executable and Linking Format (ELF) in 64-bit mode. If your application manipulates the object file format, it should support both the 64-bit ELF format and the 32-bit SOM format.

To identify the ELF format within scripts, use the HP-UX `file` command. To identify the ELF format within programs, use the *nlist64* APIs in `libelf.sl`.

Procedure Calling Conventions

The procedure calling conventions for the 64-bit PA-RISC 2.0 architecture are different. You may be impacted if your code depends on stack unwinds, uses assembly language, or passes data in and out of the kernel.

See *64-bit Runtime Architecture for PA-RISC 2.0*, URL:http://www.software.hp.com/STK/ for details.

HP-UX 64-bit Performance Considerations

Most applications should remain as 32-bit applications on HP-UX 64-bit systems. However, some applications manipulate very large data sets and are constrained by the 4GB address space limit in 32-bit mode. These applications can take advantage of the larger address space and larger physical memory limits of 64-bit systems.

Some I/O bound applications can trade off memory for disk I/O. By restructuring I/O bound applications to map larger portions of data into memory on large physical memory machines, disk I/O can be reduced. This reduction in disk I/O can improve performance because disk I/O's are more time-consuming than memory access.

Memory-constrained applications, such as large digital circuit simulations, may also benefit by transitioning to 64-bit mode. Some simulations are so large that they cannot run without major code modifications in a 32-bit address space.

What Impacts Performance in 64-bit Applications

Typical applications do not require more virtual memory than what is available in 32-bit mode. When compiled in 32-bit mode on HP-UX 64-bit platforms, these applications usually perform better than when recompiled in 64-bit mode on the same 64-bit platform. Some of the reasons for this include:

- 64-bit programs are larger. Depending on the application, the increase in the program size can increase cache and TLB misses and place greater demand on physical memory.
- 64-bit `long` division is more time-consuming than 32-bit integer division.
- 64-bit programs that use 32-bit signed integers as array indexes require additional instructions to perform sign extension each time an array is referenced.

- By default, 64-bit object modules can be placed into shared and archive libraries and used in main programs. 32-bit code must be compiled with the `+z` or `+Z` option if it is used in shared libraries.

Tuning Your 64-bit Application

Here are some ways to improve the performance of your 64-bit application:

- Avoid performing mixed 32-bit and 64-bit operations, such as adding a 32-bit data type to a 64-bit type. This operation requires the 32-bit type to be sign-extended to clear the upper 32 bits of the register.
- Avoid 64-bit `long` division whenever possible.
- Eliminate sign extension during array references. Change `unsigned int`, `int` and `signed int` variables used as array indexes to `long` variables.
- Consider compiling with the `+Onoextern` option if your 64-bit object modules are not used in a shared library.
- Consider compiling with the `+ESfic` and the `+Onoextern` options if your application is fully archive bound.

See the *HP C Online Reference* (`cc +help`) or the *HP aC++ Online Programmer's Guide* (`aCC +help`) for information on using `+Onoextern` and `+ESfic`.

4 Transitioning C and aC++ Programs to 64-bit Mode

Applications that need to take advantage of the larger address space available on HP-UX 64-bit platforms must be recompiled in 64-bit mode. Applications that do not need the larger address space, but need to perform 64-bit integer arithmetic can remain in 32-bit mode by using the `long long` 64-bit data type. This chapter provides information about when and how to port C and aC++ programs to 64-bit mode.

Step 1: Identify Programs that Need to Compile to 64-bit Mode

The decision to compile in 64-bit mode depends on your application's virtual memory requirements.

Applications requiring more than .75GB of private data or more than 1.75GB of shared memory should transition to 64-bit mode.

In 32-bit mode, the largest address space, by default, that a single process can allocate is as follows:

- .75GB for private data
- 1.75GB for shared memory

If you need to declare huge data objects, you must compile in 64-bit mode. See "Huge Data" on page 43 for more information.

Applications using the following features to extend the address space in 32-bit mode should also transition to 64-bit mode:

- `-N` linker option (`EXEC_MAGIC` flag) to access private data spaces larger than .75 GB
- `-M` option to chatr (`SHMEM_MAGIC` flag) to access shared memory larger than 1.75 GB

NOTE The `SHMEM_MAGIC` functionality will be unsupported on a future implementation of the architecture.

If your application uses data files that are greater than 2GB, you can use the large file *interface64*() routines in 32-bit mode on HP-UX 32-bit and 64-bit platforms. These routines are known as the **64()* APIs. To access files larger than 2 GB using *standard I/O* routines, you must transition to 64-bit mode.

64-bit main programs can only call 64-bit libraries. Therefore, library providers for 64-bit applications must transition their libraries to 64-bit mode.

Step 2: Identify Non-Portable Constructs

Use the HP C compiler or `lint` to find non-portable code when transitioning C programs to the LP64 data model.

A specialized `lint` tool, such as FlexeLint, can be purchased separately to help you convert 32-bit aC++ code to 64-bit code.

To turn on LP64 warnings when compiling C programs, use the `+M2` and `+DD64` (or `+M2` and `+DA2.0W`) command line options:

```
cc +DD64 +M2 -c myprog.c
```

This command line compiles in ANSI mode with HP value-added extensions (`-Ae` is the default since the HP-UX 10.30 release), turns on 64-bit porting warnings with `+M2`, suppresses linking with `-c`, and creates a 64-bit object module with `+DD64`.

`lint` provides the same 64-bit porting warnings as the HP C compiler.

NOTE `lint` additionally performs parameter checking on function calls that span compilation units.

To turn on LP64 warnings in `lint`, use the `+M2` and `+DD64` options:

```
lint +DD64 +M2 myprog.c
```

You should address all LP64 warnings issued by HP C before creating a 64-bit application.

Step 3: Make Source Code Changes

If you are already using `lint` along with the HP C `+w1` compile-line option, your port to the LP64 data model should be straightforward. When transitioning to 64-bit mode, strive to maintain a single set of source files and header files for both data models. Consider the following guidelines before porting to 64-bit mode:

- Data Truncation
  - "Avoid Assigning longs to ints" on page 61
  - "Avoid Storing Pointers in ints" on page 63
  - "Avoid Truncating Function Return Values" on page 64
  - "Use Appropriate Print Specifiers" on page 74
- Data Type Promotion
  - "Avoid Arithmetic between Signed and Unsigned Numbers" on page 62
- Pointers
  - "Avoid Pointer Arithmetic between longs and ints" on page 67
  - "Avoid Casting Pointers to ints or ints to Pointers" on page 68
  - "Avoid Storing Pointers in ints" on page 63
  - "Avoid Truncating Function Return Values" on page 64
- Structures
  - "Avoid Using Unnamed and Unqualified Bit Fields" on page 69
  - "Avoid Passing Invalid Structure References" on page 66
- Hardcoded Constants
  - "Avoid Using Literals and Masks that Assume 32 bits" on page 70
  - "Avoid Hardcoding Size of Data Types" on page 71
  - "Avoid Hardcoding Bit Shift Values" on page 72
  - "Avoid Hardcoding Constants with malloc(), memory(3), string(3)" on page 73

Avoid Assigning longs to ints

Data can be truncated when `long`s are assigned to `int`s.

To avoid this data truncation problem, change `long` to `int` assignments to assignments with the same data types.

Simple Assignment Truncation

Before:

```
int a;
long b;
. . .
a = b;  /* if b > maximum value of a 32-bit integer, then
           the result of the assignment is truncated.     */
```

Solution:

Decide if variable `b` must be `long`. If it must be `long`, make both variables `long`. Otherwise, make both variables `int`.

Bit Shift Truncation

The following `long` to `int` assignment causes an overflow condition, which leads to unexpected results:

Before:

```
#include <limits.h>
int main()
{
  long base = LONG_MAX;
  int final_result;
  final_result = base << (LONG_BIT-1); /* LONG_BIT-1 = 63 bits */
  printf("%016x\n",final_result);
}
```

The hex value for the `base` variable before the bit shift is:

```
0x 7FFF FFFF FFFF FFFF
```

The hex value of the intermediate result of the bit shift is:

```
0x 8000 0000 0000 0000
```

This 64-bit intermediate result is truncated when assigned to `final_result`. The `final_result` is:

```
0x 0000 0000 0000 0000
```

This code works in 32-bit mode since `int` and `long` are the same size. The code produces unexpected results in 64-bit mode since `final_result` is no longer big enough to hold the `long` value in `base`.

Solution:

To fix this code, make the variables `final_result` and `base` the same data type.

Diagnostic Message:

HP C generates the following LP64 migration warning for the above two examples when `+M2` and `+DD64` are enabled:

`warning 720: LP64 migration: Assignment may overflow integer` *`variable_name`*`.`

Avoid Arithmetic between Signed and Unsigned Numbers

Data is promoted differently in 64-bit mode than in 32-bit mode when `unsigned int`s are compared to `long`s, and when `int`s are compared to `unsigned long`s.

To avoid unintended data promotion problems, ANSI C programs should perform arithmetic operations and comparisons only when all operands are signed or when all operands are unsigned.

Comparison between Signed and Unsigned Numbers

The following program yields different results in 32-bit mode and 64-bit mode:

Before:

```
 1  int main()
 2  {
 3      long L = -1;
 4      unsigned int i= 1;
 5      if (L > i)
 6         printf ("L is greater than i\n");
 7      else
 8         printf ("L is not greater than i\n");
 9      return 0;
10  }
```

In 32-bit ANSI C mode, the `long` value of `-1` is promoted to an unsigned 32-bit number, making it a large positive number. In 32-bit mode, this program prints:

```
L is greater than i
```

In 64-bit ANSI C mode, both operands are promoted to signed 64-bit numbers. In 64-bit mode, this program prints:

```
L is not greater than i
```

Diagnostic Message:

HP C generates the following LP64 migration warning for this example when `+M2` and `+DD64` are enabled:

```
line 5: warning 734: LP64 migration: Different types treated as
signed for >.
```

Solution:

The code should be fixed so it produces consistent results in 32-bit and 64-bit mode. Either declare `i` as `long`:

```
long i = 1;
```

or cast `i` to a `long`:

```
if (L > (long) i);
```

Avoid Storing Pointers in ints

Pointers will be truncated in 64-bit mode if they are assigned to ints.To avoid truncation of pointers, store memory addresses in variables declared as pointers or declared with `intptr_t` (defined in `<inttypes.h>`). Store the differences between two pointers in variables declared with `ptrdiff_t` (defined in `<stddef.h>`).

Before:

```
int i;          / * 32-bit data type  * /
int j = &i;     / * This causes unexpected results. * /
```

Diagnostic Message:

HP C generates the following LP64 migration warning for this example when `+M2` and `+DD64` are enabled:

```
warning 727: LP64 migration: Initialization truncates pointer
into 32-bit integer.
```

Solution:

The solution uses the `intptr_t` type definition. This construct is portable across 32-bit and 64-bit platforms.

```
#include <inttypes.h>
int i;            / * 32-bit data type  * /
intptr_t j = &i;  / * This uses the portable typedef. * /
```

Avoid Truncating Function Return Values

The return value from function calls can be truncated if its data type is larger than or incompatible with the variable to which it is assigned.

Be aware that the C compiler assumes that functions return a value of type `int`, unless the function is properly declared.

To avoid truncation of function return values, use ANSI C function prototypes for user-defined functions and standard header files for C library functions.

Function Prototype Truncation

In the following example, `calculate_offset()` returns the difference between two pointers. In 32-bit mode, the result can be assigned to an `int` or `long`. In 64-bit mode, the result must be assigned to a 64-bit `long` or `ptrdiff_t`, as defined in `<stddef.h>`. The size of a pointer type may vary from platform to platform and from mode to mode. Therefore, using `ptrdiff_t` protects your application from different pointer sizes.

Before:

```
 1  int calculate_offset(int *base_address, int *ptr);
 2  int main()
 3  {
 4       int *base_address, *ptr;
 5       int offset;
 6       offset = calculate_offset(base_address, ptr);
 7       printf("The value of the pointer offset is \
 8              %d\n", offset);
 9       return 0;
10  }
11  int calculate_offset(int *base_address, int *ptr)
12  {
13       return (ptr - base_address);
14  }
```

Diagnostic Message:

```
line 12: warning 720: LP64 migration: Return may overflow integer.
```

Solution:

One solution is to replace the `int` return type defined in the function prototype and in `calculate_offset()` with the portable `ptrdiff_t`:

```
#include <stddef.h>
ptrdiff_t calculate_offset(int *base_address, int *ptr);
```

In 64-bit mode, this type definition is a `long`.

System Library Truncation of Return Values

The C library function `malloc()` returns a pointer. In the following example, a function prototype is not defined for `malloc()`. This leads to a run-time abort because the pointer returned by `malloc()` is truncated to an `int`.

Before:

```
1  int main ()
2  {
3    int *buffer;
4    buffer = malloc (sizeof(int));
5    *buffer = 1234;
6    printf ("The address of buffer is %p\n", &buffer );
7    printf ("The contents of buffer are %p\n", buffer );
8    printf ("The dereferenced value of buffer is \
9             %d\n", *buffer );
10   return 0;
11 }
```

At run time, this program aborts with a segmentation fault.

Diagnostic Message:

HP C generates the following LP64 migration warning for the above example when `+M2` and `+DD64` are enabled:

```
line 4: warning 724: LP64 migration: Assignment converts
default int return type to pointer "buffer".
```

Solution:

One way to fix this code is to include the `<stdlib.h>` header file, which contains the ANSI C function prototype and the K&R C function declaration for `malloc()`:

```
 1  #include <stdlib.h>
 2  int main ()
 3  {
 4    int *buffer;
 5    buffer = malloc (sizeof(int));
 6    *buffer = 1234;
```

```
 7    printf ("The address of buffer is %p\n", &buffer );
 8    printf ("The contents of buffer are %p\n", buffer );
 9    printf ("The dereferenced value of buffer is %d\n", *buffer
);
10    return 0;
11   }
```

At run time, this program now displays:

```
The address of buffer is 800003ffff8004d8
The contents of buffer are 8000000000005e60
The dereferenced value of buffer is 1234
```

Avoid Passing Invalid Structure References

HP C no longer treats a structure passed by value the same as a structure passed by reference. When calling functions that expect a pointer to a structure, be sure to explicitly pass a pointer to the structure.

Before:

```
1   struct   st_tag {int i;} a;
2   int main()   {
3                  int I;
4                  I = foo(a);
5                  return I;
6   }
7   int foo (struct st_tag *x) {
8       return (x->i);
9   }
```

Diagnostic Message:

By default, `lint` provides the following warning for this example:

```
FTN arg conflict, struct/union passed instead of ptr to
 struct/union
    foo( arg 1 )       ex.c(7)  ::  ex.c(4)
```

Solution:

```
1   struct   st_tag {int i;} *a;
2   int main()   {
3              int I;
4              I = foo(&a);       /* Pass address of struct.   */
5              return I;
6   }
             . . .
```

The solution is to pass the address of `struct a` rather than the structure itself to `foo()`.

Avoid Pointer Arithmetic between longs and ints

Dereferencing pointers using the wrong data type can yield incorrect results. In ILP32, a `long` pointer can be used to dereference an `int` value, and an `int` pointer can be used to dereference a `long` value because both values are the same length and alignment. In 64-bit mode, if a `long` value is dereferenced using an `int` pointer, only the first 32 bits of the 64-bit value will be retrieved.

Before:

```
 1  int main()
 2  {
 3      long array[5];
 4      int i;                  /* i:  index for array */
 5      int *j;                 /* j:  pointer to int  */
 6      long *k;                /* k:  pointer to long */
 7      for (i = 0; i < 4; i++)
 8          array[i]=i + 1;
 9
10      j = array + 2;          /* incrementing pointer */
11      printf ( "The address of j is %p\n", &j );
12      printf ( "The contents of j are %p\n", j );
13      printf ("The dereferenced value of j is %d\n", *j );
14
15      k = array + 2;          /* incrementing pointer */
16      printf ( "The address of k is %p\n", &k );
17      printf ( "The contents of k are %p\n", k );
18      printf ( "The dereferenced value of k is %d\n", *k );
19      return 0;
20  }
```

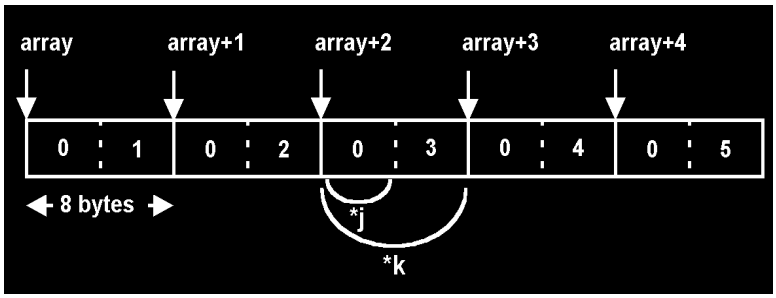
At run time, this program prints the following result:

```
The address of j is 800003ffff800510
The contents of j are 800003ffff8004f0
The dereferenced value of j is 0

The address of k is 800003ffff800518
The contents of k are 800003ffff8004f0
The dereferenced value of k is 3
```

Both `j` and `k` point to the same address, `800003ffff8004f0`. Since `j` is a pointer to an `int`, it only displays the first 4 bytes of the array, while `k` displays the entire 8 bytes as shown:

Figure 4-1



Diagnostic Message:

HP C generates the following LP64 migration warning for this example when +M2 and +DD64 are enabled:

```
line 10: warning 728: LP64 migration: Assignment converts long*
to int* “j”.
```

Solution:

The solution is to change j from an int pointer to a long pointer because the object of the pointer is an array of longs.

```
 1  int main()
 2  {
 3     long array[5];
 4     int i;          /* i:   index for array  */
 5     long *j;        /* j:   pointer to long  */
 6     long *k;        /* k:   pointer to long  */
                       . . .
```

Avoid Casting Pointers to ints or ints to Pointers

Casts made between pointers and ints will lead to unexpected results in 64-bit mode because pointers and ints are no longer the same size.

The following program aborts because of pointer truncation that results from casting a pointer as an int and storing the results in a 32-bit variable:

```
1  int main()
2  {
3     int i = 7;
4     int j;
5     int *p;
6     p = &i;
```

```
7     j = (int)p;
8     p = j;
9     j=*p;
10    return 0;
11 }
```

Diagnostic Message:

HP C generates the following LP64 migration warning for this example when +M2 and +DD64 are enabled:

```
line 7: warning 727: LP64 migration: Cast truncates pointer
into 32 bit integer.

line 8: warning 725: LP64 migration: Assignment converts 32 bit
integer to pointer "p".
```

Avoid Using Unnamed and Unqualified Bit Fields

The default run time behavior changes for programs that use unqualified or unnamed bit fields in 64-bit mode.

To avoid data alignment problems and unexpected results in programs that use bit fields, follow these guidelines:

- Use named bit fields if you expect bit fields to affect structure alignment.
- Always explicitly declare bit fields as signed or unsigned variables in ANSI C.

Before:

```
 1  struct {
 2           char foo;
 3           long : 3;  /* Unnamed bit field */
 4           short c:5;
 5  } mystruct;
 6  int main()
 7  {
 8     mystruct.c = -1;
 9     if (mystruct.c < 0)
10        printf ("The bit field is less than zero \n");
11      /* Go here in ILP32 */
12     else
13        printf ("The bit field is not less than zero \n");
14       /* Go here in LP64 */
15     printf("Size of the struct: %d\n", sizeof (mystruct));
16     return 0;
17   }
```

In 32-bit mode, this program prints:

```
The bit field is less than zero.
Size of the struct: 4
```

In 64-bit mode, this program prints:

```
The bit field is not less than zero.
Size of the struct: 2
```

Diagnostic Message:

```
line 4: warning 751: LP64 migration: Unqualified bitfields
are unsigned by default.

line 1: warning 750: LP64 migration: Unnamed, non-zero
bitfields do not affect alignment.
```

Solution:

The code should be fixed so it produces consistent results in 32-bit and 64-bit mode. In ANSI C, a portable solution is:

- name the bit field,
- declare signed types, and
- change the `long` variable to an `int`.

```
struct {
        char foo;
        signed int b: 3;
        signed short c: 5;
             . . .
```

Avoid Using Literals and Masks that Assume 32 bits

Programs that assign hardcoded numeric constants to longs in 32-bit mode may get different results in 64-bit mode if they assume a `long` is 32 bits. In 64-bit mode, this assumption is incorrect.

To avoid setting incorrect bit mask and hexadecimal values, use pre-processor directives with the the `__LP64__` predefined type to generate platform specific code. Or, consider using the complement (~) operator.

Using the __LP64__ Predefined Macro and Complement (~) Operator

In the following example, the programmer wants to set all but the last 4 bits of the `long` value to 1. This code works in 32-bit mode. However, in 64-bit mode, it sets the leftmost 32 bits to 0, the next 28 bits to 1, and the last 4 bits to 0.

Before:

```
long L = 0xFFFFFFF0;   / * 32-bit hex constant '1111...0000' * /
```

Solution 1:

```
#ifdef __LP64__
   long L = 0xFFFFFFFFFFFFFFF0;
#else
   long L = 0xFFFFFFF0;
#endif
```

The solution `ifdef`s the `long` to the correct 64-bit value.

Solution 2:

```
long L = ~0xFL;   /* Sets the rightmost 4 bits to 0.  */
```

The solution uses the complement (~) operator to reverse each bit in the operand. The suffix `L` is required for the `long` value so that HP C uses the correct length for the constant.

Avoid Hardcoding Size of Data Types

`grep` for hardcoded system constants that represent sizes of data types.

Here are some common 32-bit system constants:

`4`	number of bytes in a 32-bit pointer
`32`	number of bits in a 32-bit pointer
`2147483647`	maximum value of 32-bit signed integer
`-2147483648`	minimum value of 32-bit signed integer
`4294967295`	maximum value of 32-bit unsigned integer
`0x7fffffff`	maximum value of 32-bit signed integer in hexadecimal format
`0xffffffff`	maximum value of 32-bit unsigned integer in hexadecimal format

`0x80000000`	minimum value of 32-bit signed integer in hexadecimal format

Instead of hardcoding values, use the macro values in include files, such as `<limits.h>` and `<inttypes.h>`. Examples of macros in `<limits.h>` include `INT_MAX`, `LONG_MAX`, and `CHAR_BIT`. See the man page for *limits*(5) for more information. A better solution is to use the `sizeof()` operator.

Before:

```
offset = n * 4;            /*  Assumes long is 32 bits.        */
```

Solution 1:

```
offset = n * sizeof(long); /*  Works if long is 32 bits
                               or 64 bits.                     */
```

Solution 2:

```
long val;
offset = n * sizeof(val); /*  Optimal!  The algorithm adjusts
                              to current size of val.          */
```

Avoid Hardcoding Bit Shift Values

Check for hardcoded values in bit shift operations. The following examples only get expected results in 32-bit mode because they all hardcode a shift range of 32 bits:

Before:

```
unsigned long n, newexp;
n = n >> (32 - newexp); /* Not portable! Assumes long is 32 bits.*/
```

Solution:

```
#include <limits.h>
unsigned long n, newexp;
n = n >> (LONG_BIT - newexp);
```

Uses the portable `LONG_BIT` from `<limits.h>`.

Before:

Here is an example that turns on the high bit (signed or leftmost bit):

```
long n;
n = 1 << 32-1;  /* Not portable!  Assumes long is 32 bits. */
```

Solution 1:

```
#include <limits.h>
long n;
n = ~ LONG_MAX;       /* This is portable. */
```

Solution 2:

```
#include <limits.h>
long n;
n = 1L << LONG_BIT-1; /* This is portable. */
```

Avoid Hardcoding Constants with malloc(), memory(3), string(3)

Check for hardcoded constants in programs that use the *memory*(3C) and *string*(3C) family of functions. Unfortunately, hardcoding constants is common when the following routines are used:

- *malloc*(3C) family of memory allocators — `realloc()`, `calloc()`, `valloc()`, `alloc()`
- *memory*(3C) family of memory operators — `memccpy()`, `memcmp()`, `memcpy()`, `memmove()`
- *string*(3C) family of string operators — `strcpy()`, `strcat()`, `strncmp()`, `strncpy()`

Before:

```
#include <stdlib.h>
#define BSIZE 4096    /* Buffer size */
char **pointer_buffer;
pointer_buffer=(char **)malloc(BSIZE*4);  /* Assumes pointer
                                             is 4 bytes.     */
```

The `pointer_buffer` is a buffer of pointers. The `malloc()` function attempts to allocate enough space for this buffer. This code does not allocate enough space for `pointer_buffer` because the size of a pointer is 8 bytes in 64-bit mode. Silent data corruption or a core dump can happen when the upper bound of the buffer is reached.

Diagnostic Message:

None.

Solution:

```
#include BSIZE 4096     /* Buffer size */
#include <stdlib.h>
void **pointer_buffer;
pointer_buffer=(void **)malloc(BSIZE*(sizeof(void *)));
```

The `sizeof(void *)` operator replaces the hardcoding of the size of a pointer.

Use Appropriate Print Specifiers

When using `varargs`, `stdarg`, or variable argument interfaces such as `printf()`, you must ensure that the algorithms that use the variable parameters are consistent with the 64-bit data model.

Before:

```
long value;
printf("value = %d\n", value);
```

The `d` specifier prints a 32-bit integer.

After:

```
long value;
printf("value = %ld\n", value);
```

The `ld` specifier prints a 64-bit integer in 64-mode and a 32-bit integer in 32-bit mode.

Before:

```
long offset;
printf("offset =%x\n", offset);
```

The `x` specifier prints a 32-bit hexadecimal number.

After:

```
#include <inttypes.h>
int64_t offset;
printf("offset =" %PRIx64 "\n", offset);
```

The `PRIx64` macro prints a 64-bit integer in both modes.

Step 4: Compile in 64-bit Mode

To compile in 64-bit mode, update makefiles with the appropriate options:

`+DD64`	Compiles C programs in 64-bit mode. Use this option for compatibility with future architectures.
`+DA2.0W`	Compiles aC++,C, or Fortran 90 programs in 64-bit mode.

If you are building 64-bit applications on HP-UX 11.0 32-bit systems, be sure to request and verify that 64-bit versions of system libraries are installed. For example, verify that `/usr/lib/pa20_64/libc.sl` is installed.

64-bit applications only run on 64-bit systems.

64-bit main programs and object files can only link with 64-bit object files and 64-bit versions of shared or archive libraries.

5 Writing Portable Code

This chapter provides information on coding practices for developing code that is portable between HP-UX 32-bit, 64-bit, and other UNIX systems. It includes how to use industry-standard header files and HP-UX extensions that isolate platform-specific code.

The following topics are included:

- Making code 64-bit clean
- Using integral types defined in `<inttypes.h>`
- Guidelines for using `<inttypes.h>`
- Using *pstat(2)* instead of `/dev/kmem`
- Getting configurable system information
- Isolating system-specific code
- Using system-specific include files

Making Code 64-bit Clean

If your application is targeted for execution on both ILP32 and LP64, it should be 64-bit clean. Tools to help make your code 64-bit clean include:

- using `lint`,
- compiling in ANSI C or aC++, and
- using the portable header file `<inttypes.h>`.

To make code 64-bit clean, follow these guidelines:

- Use the same source code and header files for both 32-bit and 64-bit programs.
- Use appropriate data types consistently and strictly.

  For example, use `off_t` consistently for file offsets and `fpos_t` for file positions.
- Use integral types in `<inttypes.h>`, where applicable, instead of `int` and `long`
- Use fixed/scalable width integral types, algorithms, and masks as appropriate.

  Fixed types remain a consistent size on 32-bit and 64-bit platforms. For example, use `int32_t`, defined in `<inttypes.h>`, if `int`s and `long`s should be 32 bits in your application. Scalable types can grow and scale without modification to future architectures.
- Perform boundary checking on integral type ranges.
- Update 64-bit code in cases where 32-bit and 64-bit processes share the same memory segment.

Using Integral Types Defined in <inttypes.h>

The `<inttypes.h>` header files provide the following features that help you write portable code:

- integral type definitions
- macros for constants
- macros for `printf()`/`scanf()` specifiers

When you use `<inttypes.h>`, you can maintain one set of source files for both data models. Using the `typedef`s and constants included in these header files protects your code from underlying data model changes. It also reduces the need to `#ifdef` platform-specific code, which improves the readability of your source code.

There are two `<inttypes.h>` header files. The following information is included in `<sys/_inttypes.h>`:

- basic integral data types for 8, 16, 32, and 64 bits
- macros that can create constants of a specific size and sign

The following information is included in `<inttypes.h>`:

- data types for pointers
- data types for the most efficient integer types
- data types for the largest integer types
- data types for the smallest integral types of at least 8, 16, 32, and 64 bits

Integer Data Types with Consistent Lengths

The basic types in `<sys/_inttypes.h>` have consistent lengths on UNIX 32-bit and 64-bit platforms. Use these basic data types *instead* of the standard C language types whenever you want data types to remain the same size across different architectures.

Table 5-1 shows the basic integral types in `<sys/_inttypes.h>`:

Table 5-1 C Basic Integer Types in <sys/_inttypes.h>

Type Definition Name	Description
int8_t	8-bit signed integer
uint8_t	8-bit unsigned integer
int16_t	16-bit signed integer
uint16_t	16-bit unsigned integer
int32_t	32-bit signed integer
uint32_t	32-bit unsigned integer
int64_t	64-bit signed integer
uint64_t	64-bit unsigned integer

The pointer data types are signed and unsigned integer data types that are large enough to hold a pointer. A pointer can be moved to or from these data types without corruption.

Table 5-2 Pointer Types in <inttypes.h>

Type Definition Name	32-bit Mode	64-bit Mode
intptr_t	32-bit signed integer	64-bit signed integer
uintptr_t	32-bit unsigned integer	64-bit unsigned integer

Using Integer Data Types with Consistent Lengths

One way to improve portability of programs that require integral data items to be 32-bits or 64-bits long, regardless of the hardware platform, is to `#include` the `<inttypes.h>` header file, and to make the following substitutions:

Table 5-3

Instead of	Use
`short`	`int16_t`
`unsigned short`	`uint16_t`
`int`	`int32_t`
`unsigned int`	`uint32_t`
`long`	`int32_t` or `int64_t`
`unsigned long`	`uint32_t` or `unint64_t`
`long long`	`int64_t`
`unsigned long long`	`uint64_t`

Excerpts from <sys/_inttypes.h>

Here are some typedefs in `<sys/_inttypes.h>` that conform to the XPG5 standard:

```
typedef int int32_t;      /* 32-bit signed integral type   */
typedef int64_t intmax_t;  /* largest signed integral type */
typedef unsigned long uint64_t; /* 64-bit unsigned integral type
*/
```

Here are some HP extensions:

```
#define INT32_C(_c)     /* Create a 32-bit signed constant     */
#define UINT32_C(_c)    /* Create a 32-bit unsigned constant   */

#define UINT64_MAX /* If this macro tests false, the unsigned
                      64-bit data type is not supported on
                      the platform.                           */

#define PRId64 /* printf specifier for a 64-bit integral value */
#define SCNd64 /* scanf specifier for a 64-bit integral value  */
```

Examples Using Consistent Length Data Types and Macros

The following code uses constructs that are not 64-bit clean:

Before:

```
#include <limits.h>
#include <stdio.h>
          . . .
long value;
long mask = 1L << ((sizeof(long) * CHAR_BIT)-1);
scanf("%ld", &value);
if (value = 0x7fffffff)    /* Test for max 32-bit value. */
   printf("Number is %ld\n", value);
if (mask & value)          /* Handle bad value.          */
          . . .
```

The after code has been made 64-bit clean by using `<inttypes.h>`:

After:

```
#include <limits.h>
#include <stdio.h>
#include <inttypes.h>
          . . .
int32_t value;
int32_t mask = INT32_C(1)<<((sizeof(int32_t) * CHAR_BIT)-1);
scanf("%" SCNd32, &value);
if (value = INT32_MAX)  /* Test for max 32-bit value.*/
    printf("Maximum 32-bit signed integer is \
             %" PRId32 "\n", value);
if (mask & value)       /* Handle bad value.         */
        . . .
```

Example 2: Formatted I/O using printf(3S)

Use the `PRIx`*n* constants defined in `<inttypes.h>` to select the correct `printf()` formatting option for the data types defined in `<inttypes.h>`. For example:

Before:

```
long result_value;
printf ("Result = %lx\n", result_value);
```

After:

```
#include <inttypes.h>
int32_t result_value;
printf ("Result = %" PRIx32 "\n", result_value);
```

The *before* code prints a 32-bit number on 32-bit platforms and a 64-bit number on a 64-bit platform. The *after* code prints a 32-bit number, regardless of the platform. The macro `PRIx32` from `<inttypes.h>` is used instead of the literal `lx`. `PRIx32` is then defined as the appropriate formatting symbol.

intfast Data Types with Defined Minimum Sizes

When execution speed of programs is important, use the `intfast`*x*`_t` data types in `<inttypes.h>`. You select data types based on the minimum integer size required by your application. The `<inttypes.h>` header file then maps the `intfast` data types to the fastest appropriate data type for the target hardware platform. These data types include:

`intfast_t`	fastest signed integral data type of at least 16-bits
`intfast8_t`	fastest signed integral data type at least 8-bits long
`uintfast8_t`	fastest unsigned integral data type at least 8-bits long
`intfast16_t`	fastest signed integral data type at least 16-bits long
`uintfast16_t`	fastest unsigned integral data type at least 16-bits long
`intfast32_t`	fastest signed integral data type at least 32-bits long
`uintfast32_t`	fastest unsigned integral data type at least 32-bits long

Example Using intfast*n*_t Data Types

Suppose you need a loop to execute as fast as possible on a target platform. You could change the following:

Before:

```
int i, j;
j = 0;
for  (i = 1; i<=127; i++)
{
 j = j+i;
}
```

to use the most efficient and portable integer data types as follows:

After:

```
intfast8_t i;
intfast16_t j = 0;
for (i =1; i<=INT8_MAX; i++)
{
 j = j+i;
}
```

Guidelines for Using <inttypes.h>

When porting to 64-bit mode, you should only use 64-bit data types when they are required. This data type may not be processed efficiently on some platforms. Programs may run slower and executable file sizes may be bigger.

Here are additional guidelines:

- Use `int32_t` for integers that must be 32 bits in length on 64-bit systems in order for programs to work correctly.

  Because `<inttypes.h>` is available on both 32-bit and 64-bit HP-UX platforms, `int32_t` works in both environments.

- Use the smallest integral type possible in order to control the size of the application.
- Use the `intfast`*n*`_t` data types for counters and counter loops that need to be fast due to frequent expression evaluations (such as incrementing).
- Use the constant that matches the integer type definition.

  For example, use the constant `UINT64_FAST_MAX` with `uint_fast64_t`. Do not use `INT_MAX` or `UINT_MAX` with `uint_fast64_t`.

- Use `intmax_t` or `uintmax_t` for items that must be the largest available integral type as specified by the compiler.

  The sizes of these data types may change in a future release, but they will always be the largest integral type supported on the platform, regardless of possible performance implications.

- Limit the use of *x*`64_t` data types:

  - These types cannot automatically take advantage of potentially higher limits for future integral sizes.
  - Older 32-bit systems may not have a 64-bit integral data type. Therefore, `int64_t` and `uint64_t` data types may need to be protected by `#ifdef`s if the source code is shared between 64-bit and older 32-bit systems.

- Convert `long long` to `int64_t`. Convert `unsigned long long` to `uint64_t`.
- End user APIs reflect the data types defined by standards such as X/Open, POSIX.2, or legacy HP definitions. They will not cause source level incompatibilities.

  The function prototype looks the same in any integral data type model. Therefore, your application will be protected when there are changes to the underlying size of data types. For example, the function prototype for `lseek()`:

  ```
  off_t lseek(int fildes, off_t offset, int whence);
  int fseek(FILE *stream, long int offset, int whence);
  ```

  looks the same on a 32-bit or 64-bit system.
- Use the same type definition names supported by the API definition. For example, use `off_t` in your code offsets. Do not assume `off_t` is an `int`, `long`, or any other data type.
- Data declarations related to API parameters or return values should be of the same consistent data definition as defined by the function prototype.
- Integral types (for example, `long`) that must be 32 bits on a 32-bit system and 64 bits on a 64-bit system can be left as `long`. These types will automatically be declared with the correct size.
- Use scalable masks with scalable `typedef`s and fixed size masks with fixed size `typedef`s. (See “Using Portable Bit Masks” on page 87 for examples.)

Using portal.h

The `portal.h` header file helps you to write code that is portable across HP-UX 32-bit and 64-bit systems. This header file contains macros for setting bit masks and sign bits of data types defined in `<inttypes.h>` and for convenience it includes the header files `<limits.h>` and `<inttypes.h>`.

To `#include` this header file, type:

```
#include <sys/portal.h>
```

Examples of some macros contained in `portal.h` follow:

- `SET_MASK_BIT`(*bit_num*, *data_type*) — Creates a mask that has one bit set.

  For example:

  ```
  SET_MASK_BIT(SIGN_BIT(int64_t), int64_t)
  ```

  turns on the high bit in a 64-bit integer.

- `SIGN_BIT` (*data_type*) — Returns the bit position of the sign bit for the specified type.

  For example:

  ```
  SIGN_BIT(int32_t)
  ```

  returns the position of the sign bit in a 32-bit integer.

- `SIGN_EXTEND` (*value*, *old_type*, *new_type*) — Sign extends from one data type to another.

  For example:

  ```
  char c;
  int64_t i;
  i = SIGN_EXTEND(c, char, int64_t);
  ```

  converts the 8-bit integer stored in a `char` data type to a 64-bit integer and correctly extends the sign.

For additional information, see the man page for *portal.h*(5).

Using Portable Bit Masks

Use scalable masks with scalable `typedef`s. Scalable types, such as `int` and `long`, may be different sizes on different platforms. For example, use the portable bit mask construct:

- Solution 1:

```
#include <inttypes.h>
#ifdef __LP64__
   int64_t node_id=0xfffffffffffffffc;
#else
   int32_t node_id=0xfffffffc;
```

or:

- Solution 2:

```
#include <inttypes.h>
intmax_t node_id = ~0x3;
```

instead of the non-portable construct:

```
long node_id =  0xfffffffc;
```

When compiled on an HP-UX 32-bit platform, the first 2 constructs above result in the intended value:

```
1111 1111 1111 1111 1111 1111 1111 1100
```

However, when compiled on an HP-UX 64-bit platform, the first and second construct produce the correct binary value:

```
1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111 1111 1111 1100
```

while the third construct generates an incorrect bit mask value:

```
0000 0000 0000 0000 0000 0000 0000 0000
1111 1111 1111 1111 1111 1111 1111 1100
```

Use fixed size masks with fixed size type definitions. For example, if you want a 32-bit mask, use the portable construct:

```
#include <inttypes.h>
int32_t intmask = INT32_MAX;
```

instead of:

```
long intmask = 0x7fffffff;
```

Using pstat(2) instead of /dev/kmem

Avoid writing programs that read system files such as `/dev/kmem` directly. These files may have different formats on different systems. Instead, use the *pstat*(2) routines to get system information. These routines were first introduced in the HP-UX 10.0 release. Using *pstat*(2), information can be retrieved about the following:

- memory — static, dynamic, shared, virtual
- system CPU processors
- system processes and messaging
- disks, swap space, and file systems
- semaphores

NOTE The *pstat*(2) routines are HP-specific routines and are not available on other vendor platforms.

For More Information: See the *pstat(2)* man page.

Getting Configurable System Information

Instead of hardcoding system values, use any of the portable alternatives:

- Use `sysconf(2)`, `confstr(3C)`, and `getconf(1)` to get configurable system information, such as determining if the underlying operating system is 64 bits or 32 bits.
- Use `pathconf(2)` to get configurable pathnames for files and directory values.
- Use `<limits.h>` to get static system limits.

Beginning with the 11.0 release, `sysconf()` returns additional information:

- the processor implementation
- the operating system execution mode — 32 bit or 64 bit
- whether the system is capable of running a 64-bit operating system

Additionally, the `confstr()` system call returns the appropriate compiler options, libraries, and lint options to build and check an application in a 32-bit or 64-bit programming environment.

For More Information:

- See the man page for *limits*(5) for details on macros available in `<limits.h>`.
- See the man pages for *sysconf*(2), *confstr*(3C), *getconf*(1), and *pathconf*(2).

Isolating System-Specific Code

Use conditional directives to isolate sections of platform or release-specific code. This is useful when the number of lines of release or platform-specific source code is small in relation to the number of lines of common code. You could, for example, encapsulate code that is different between the 32-bit and 64-bit HP-UX with conditional directives.

The following example uses a #define macro within a conditional directive:

```
#ifdef __LP64__
   . . .     /* LP64 specific code goes here */
#else
   . . .     /* ILP32 specific code goes here */
#endif
   . . .     /* Code common to both platforms  goes here */
```

Using System-Specific Include Files

Sometimes platform or release-specific source code differences are large. These differences can be isolated in different `#include` files and then be referenced by conditional directives.

Here is an example of including 32-bit or 64-bit type definitions depending on the word size of the machine:

```
#include <limits.h>
#if WORD_BIT > 32
      . . .         /* integers must be 64-bit */
#   include "typedefs_64.h"
#else
      . . .         /* integers must be 32-bit */
#   include "typedefs_32.h"
#endif              /* 32-bit environment */
```

Glossary

archive library A library created by the ar command, which contains one or more object modules. By convention, archive library file names end with .a. Compare with “shared library.”

breadth-first search order The dependent library search algorithm used when linking and loading 64-bit applications.

bss segment A segment of memory in which uninitialized data is stored. Compare with “hbss segment,” “text segment” and “data segment.” For details, refer to a.out(4).

data alignment Refers to the way a system or language aligns data structures in memory.

data segment A segment of memory containing a program’s initialized data. Compare with “bss segment,” “hbss segment” and “text segment”. For details, refer to a.out(4).

data type promotion The conversion of operands with different data types to compatible types for comparison and arithmetic operations.

dependency Occurs when a shared library depends on other libraries — that is, when the shared library was built (with `ld -b...`), other libraries were specified on the command line. See also "dependent library."

dependent library A library that was specified on the command line when building a shared library (with `ld -b...`). See "dependency."

depth-first search order The dependent library search algorithm used when linking and loading 32-bit applications.

dynamic loader Code that attaches a shared library to a program. See *dld.sl*(5).

dynamic path searching The process that allows the location of shared libraries to be specified at run time.

external reference A reference to a symbol defined outside an object file.

hbss segment A segment of memory in which uninitialized huge data is stored. Compare with “bss segment,” “text segment” and “data segment.” For details, refer to a.out(4).

Glossary

huge data In general, any data object larger than can be represented on a 32-bit system; more specifically, any data object greater than a specified threshold that is placed in an hbss segment.

ILP32 The HP-UX 32-bit data model. In this model, `int`s, `long`s and pointers are 32 bits in size.

LP64 The HP-UX 64-bit data model. In this model, `long`s and pointers are 64 bits in size and `int`s are 32 bits.

link order The order in which object files and libraries are specified on the linker command line.

magic number A number that identifies how an executable file should be loaded. Possible values are `SHARE_MAGIC`, `DEMAND_MAGIC`, and `EXEC_MAGIC`. Refer to *magic*(4) for details.

object code See relocatable object code.

object file A file containing machine language instructions and data in a form that the linker can use to create an executable program.

object module A file containing machine language code and data in a form that the linker can use to create an executable program or shared library.

pipe An input/output channel intended for use between two processes: One process writes into the pipe, while the other reads.

pragma A C directive for controlling the compilation of source.

relocatable object code

Machine code that is generated by compilers and assemblers. It is relocatable in the sense that it does not contain actual addresses; instead, it contains symbols corresponding to actual addresses. The linker decides where to place these symbols in virtual memory, and changes the symbols to absolute virtual addresses.

shared executable An `a.out` file whose text segment is shareable by multiple processes.

shared library A library, created by the `ld` command, which contains one or more PIC object

modules. Shared library file names end with `.sl`. Compare with "archive library."

standard input/output library

A collection of routines that provide efficient and portable input/output services for most C programs.

text segment A segment of read-only memory in which a program's machine language instructions are typically stored. Compare with "bss segment" and "data segment." For details, refer to *a.out*(4).

Glossary

Index

Index

Index

Index

Index

X