overview

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

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 the following table:

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

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.

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

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.

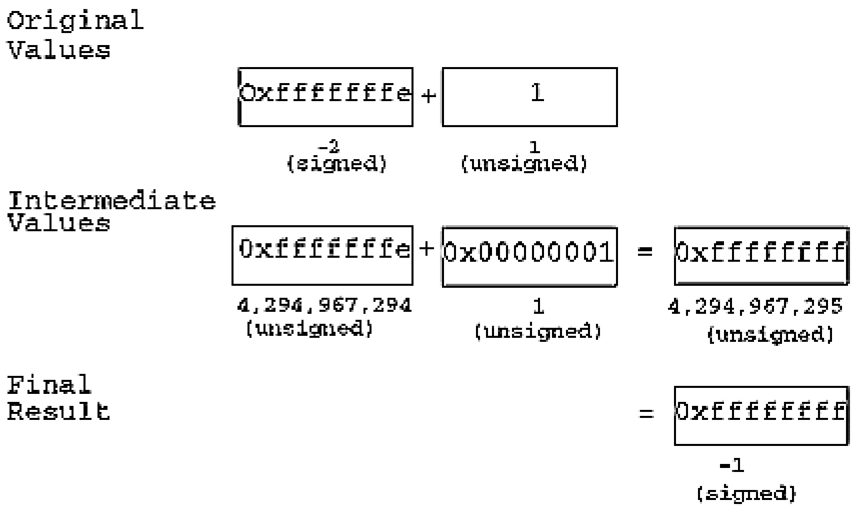
data type promotion

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*.

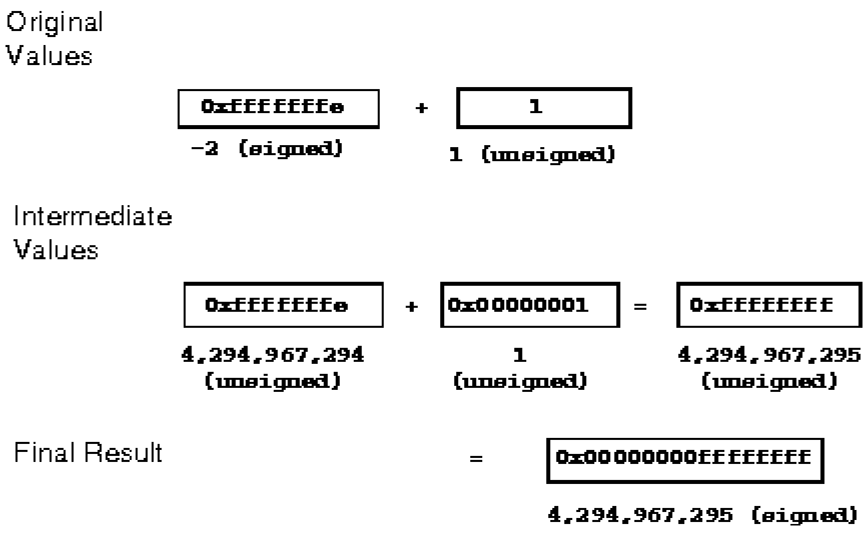
Certain data type promotions result in signed numbers being treated as unsigned numbers. When this happens, you can occasionally get unexpected results. For example:

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

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



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:



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.

This table shows the data alignment for C data types:

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	8-byte
`struct`	depends on members *(1)*	depends on members*(1)*	depends on members*(1)*	depends on members*(1)*
`enum`	4	4-byte	4	4-byte

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

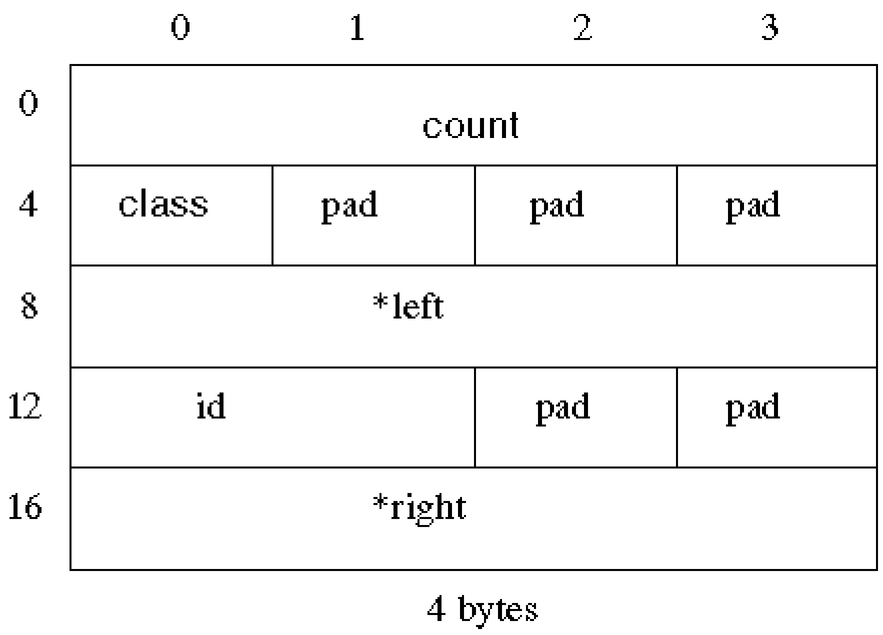
structure member alignment

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;
}
```

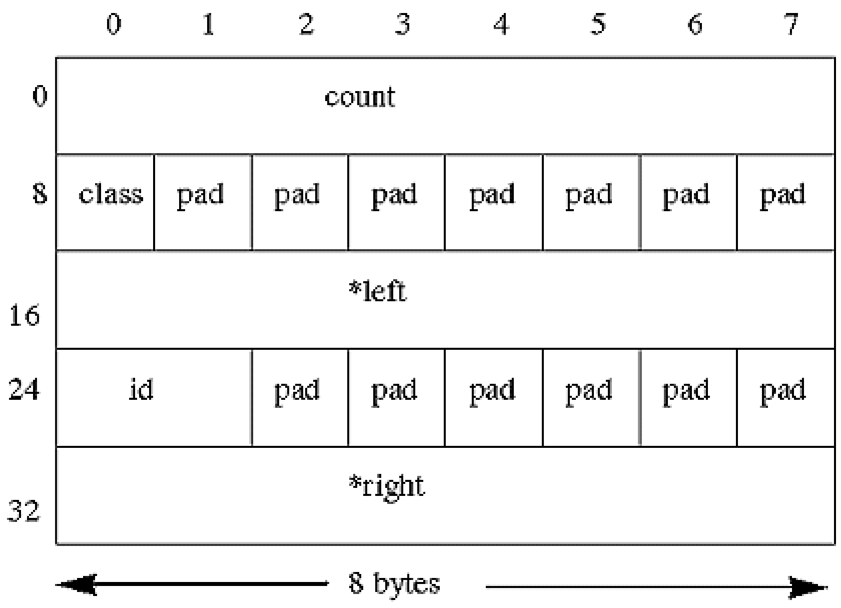
The `tnode` structure is aligned according to the alignment shown in ILP32 and LP64 Data Alignment. The following diagram shows the alignment for `tnode` in ILP32:



ILP32 Alignment of struct tnode

In ILP32, this data structure contains 20 bytes.

The following diagram shows the alignment for the `tnode` structure in LP64.



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 on different platforms.

For information on how to create portable data structures, see Writing Portable Code.

constants

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

constant	ANSI C ILP32	ANSI C LP64
`0x7fffffff`	`int`	`int`
`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:

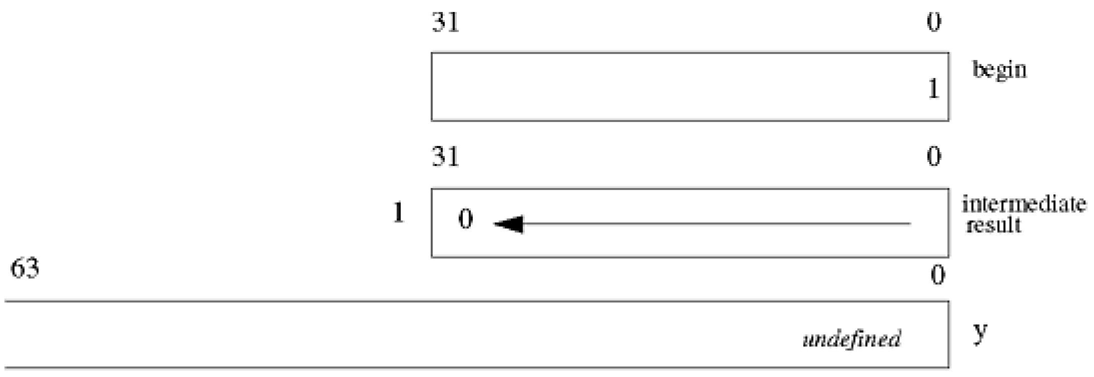
`a = b` *operation* `c`

the data type used for the intermediate results 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;
/* Overflows under both data models. */
y = (1 << 32);
```

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:



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:

```
/* 2^32 in LP64. Overflows in ILP32. */
```

```
y = (1L << 32);
```

enumerated types

In LP64, enumerated types are signed only if one or more of the enumeration constants defined for that type are 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 2.0 calling conventions You may also want to take advantage of the new instructions for improved performance. If you are transitioning to IPF platforms, of course, the assembly language is different and will need to be rewritten. You can find more information on IPF assembly language at IA-64 Instruction Set Architecture Guide.

The following summarizes items that may need adjustments for 64-bit PA:

- 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.
- 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 changed.

object file format

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

HP PA 2.0-based systems (64-bit systems) use the SOM object file format in 32-bit mode and the industry standard Executable and Linking Format (ELF) in 64-bit mode. Applications that are knowledgeable about the object file format need to support ELF for 64 bits and SOM for 32 bits.

Scripts can identify the ELF format by using the HP-UX `file` command. Programs can identify the ELF format by using the nlist64 APIs in `libelf.sl`.

procedure calling conventions

The procedure calling conventions are changing for the 64-bit PA 2.0 architecture. Code that deals with stack unwinds, assembly language, and passing data in and out of the kernel may be impacted.

See 64-Bit Run-time Architecture for PA-RISC 2.0 (`.pdf`) for details.

HP-UX 64-bit performance considerations

Most applications can remain in 32-bit mode on HP-UX 64-bit systems. However, some applications, which manipulate very large data sets, 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 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 of these simulations have grown to the point where they cannot run without major code modifications in a 32-bit address space.

what impacts performance in 64-bit mode

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 a binary executable (`a.out`, not `.o`, `.a` or `.sl`).

see also

For additional information on C or C++, see:
» HP aC++ Online Programmer's Guide
» HP C/HP-UX Release Notes
» HP aC++ Release Notes

For additional information on Fortran, see:
» HP Fortran 90 Release Notes

For additional information on assembly language changes, see:
» Assembler Reference Manual
» IA-64 Instruction Set Architecture Guide
» 64-Bit Run-time Architecture for PA-RISC 2.0 (`.pdf`)

For additional information on linkers and libraries, see:
» *HP-UX Linker and Libraries User's Guide*

For addition information on writing portable code, see:
» Writing Portable Code
» IPF Software Developer's Portability Checklist
» IPF Software Developer's Performance Checklist