Oracle® Solaris 64-bit Developer's Guide

Part No: E61689
March 2019

ORACLE

Oracle Solaris 64-bit Developer's Guide

Part No: E61689

Référence: E61689

Accès aux services de support Oracle

Les clients Oracle qui ont souscrit un contrat de support ont accès au support électronique via My Oracle Support. Pour plus d'informations, visitez le site `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` ou le site `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` si vous êtes malentendant.

Contents

Using This Documentation

- **Overview** – For application developers who are converting 32-bit applications to 64-bit and developing 64-bit applications on Oracle Solaris. It describes the 32-bit and 64-bit application environments and describes some of the utilities and commands available for developing 64-bit applications.

  Oracle Solaris 11.3 is a 64-bit only operating system and provides an environment to build and run 64-bit applications that can use large files and large virtual address spaces. At the same time, to maintain backward compatibility, this release supports 32-bit applications.

  The Oracle Solaris OS supports systems that use the SPARC and x86 families of processor architectures: UltraSPARC, SPARC64, AMD64, Pentium, and Xeon EM64T. For a list of all the supported systems, see Oracle Solaris Hardware Compatibility List.

  The major differences between the 32-bit and the 64-bit application development environments are as follows.

  - 32-bit applications are based on the ILP32 data model, where `int`, `long`, and pointers are 32-bit.
  - 64-bit applications are based on the LP64 data model, where `long` and pointers are 64 bits and the other fundamental types are the same as in ILP32 data model.
  - In a 64-bit development environment, the large file interface is no longer required. The large file interface enables 32-bit programs to handle files that are larger than 2GB.
  - 32-bit `time_t` can only handle dates up to January 2038. However 64-bit `time_t` can handle dates for billion years into the future.

  You might want to convert your application from 32-bit to 64-bit if your application has one or more of the following requirements:

  - Needs more than 4 gigabytes of virtual address space
  - Reads and interprets kernel memory through use of the `libkvm` library, and `/dev/mem`, or `/dev/kmem` files
  - Uses the `/proc` interface to debug 64-bit processes
  - Uses a library that has only a 64-bit version
  - Needs full 64-bit registers to do efficient 64-bit arithmetic
  - Use dates beyond January 2038

Specific interoperability issues can also require code changes. For example, if your application uses files that are larger than 2 gigabytes, you might want to convert the application to 64-bit.

In some cases, you might want to convert applications to 64-bit for performance reasons. For example, you might need the 64-bit registers to do efficient 64-bit arithmetic or you might want to take advantage of other performance improvements that a 64-bit instruction set provides.

Note - In this document the term "x86" refers to 64-bit and 32-bit systems manufactured using processors compatible with the AMD64 or Intel Xeon/Pentium product families.

- **Audience** – Application developers who intend to convert 32-bit applications to 64-bit applications and develop 64-bit applications on Oracle Solaris 11 and later versions.
- **Required knowledge** – Experience in developing applications in C and an understanding of the 32-bit and the 64-bit architectures.

Product Documentation Library

Documentation and resources for this product and related products are available at http://www.oracle.com/pls/topic/lookup?ctx=E53394-01.

Feedback

Provide feedback about this documentation at http://www.oracle.com/goto/docfeedback.

♦ ♦ ♦ CHAPTER 1

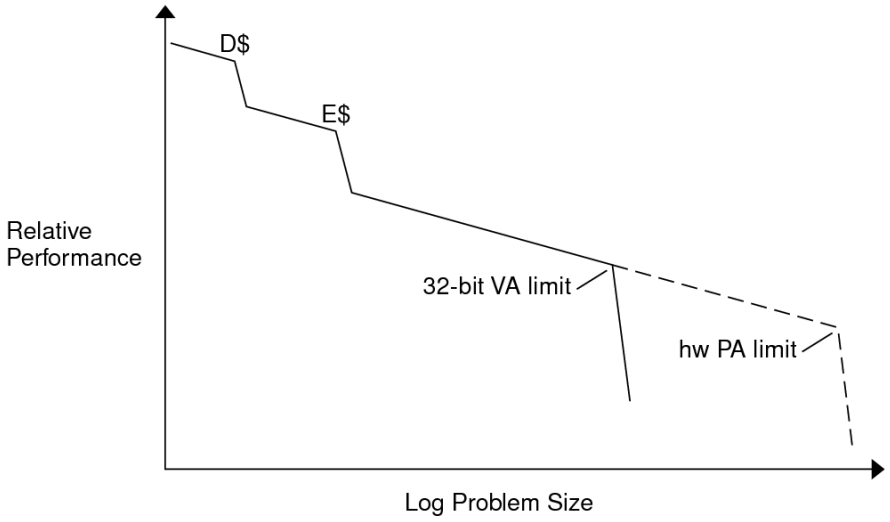
64-Bit Computing

As applications continue to become more functional and more complex, and as data sets grow in size, the address space requirements also continue to grow. Now a days most of the applications are 64-bit and the 4 Gigabyte address space provided by 32-bit systems is no longer sufficient.

1.1 Getting Past the 4 Gigabyte Barrier

The diagram in Figure 1, “Typical Performance and Problem Size Curve,” on page 12 shows how an application undergoes a significant drop in performance with increase in problem size on a system with a large amount of physical memory. For very small problem sizes, the entire program can fit in the data cache (`D$`) or the external cache (`E$`). But eventually, the program's data area becomes large enough that the program fills the entire 4 Gigabyte virtual address space of a 32-bit application.

FIGURE 1 Typical Performance and Problem Size Curve



Beyond the 32-bit virtual address limit, applications programmers can still handle large problem sizes. Usually, applications that exceed the 32-bit virtual address limit split the application data set between primary memory and secondary memory, for example, onto a disk. Unfortunately, transferring data to and from a disk drive takes a longer time, in orders of magnitude, than memory-to-memory transfers.

Today, many servers can handle Terabytes of physical memory. A single 32-bit application cannot directly address more than 4 Gigabytes at a time. However, a 64-bit application can use the 64-bit virtual address space capability to allow up to 18 Exabytes (1 Exabyte is approximately 10^{18} bytes) to be directly addressed. Thus, larger problems can be handled directly in primary memory. If the application is multithreaded and scalable, more processors can be added to the system to speed up the application even further. Such applications become limited only by the amount of physical memory in the system.

For a wide range of applications, the ability to handle larger problems directly in primary memory *is* the major performance benefit of 64-bit machines. Some of the advantages are as follows:

- A greater proportion of a database can live in primary memory.
- Larger CAD/CAE models and simulations can fit in primary memory.
- Larger scientific computing problems can fit in primary memory.
- Web caches can hold more in memory, reducing latency.

1.2 Beyond Large Address Spaces

Other compelling reasons why you might want to create 64-bit applications are as follows:

- You can perform a lot of computation on 64-bit integer values that use the wide data paths of a 64-bit processor to gain performance.
- Several system interfaces have been enhanced, or limitations removed, because the underlying data types that underpin those interfaces have become larger. For example, 32-bit `time_t` can only handle dates up to January 2038, while 64-bit `time_t` can handle dates for billion years into the future.
- You can obtain the performance benefits of the 64-bit instruction set, such as improved calling conventions and full use of the register set.
- You can increase the security by using ASLR, which helps you to create more random, harder to guess address spaces. For more information, see "Randomizing the Layout of the Address Space" in *Securing Systems and Attached Devices in Oracle Solaris 11.3* and "Security Extensions Framework" in *Developer's Guide to Oracle Solaris 11.3 Security*.

♦ ♦ ♦ CHAPTER 2

When to Use 64-Bit for Applications

For application developers using Oracle Solaris, the major difference between the 64-bit and 32-bit operating environments is the C data type model used. The 64-bit version uses the LP64 data model where `long` and pointers are 64-bit. All other fundamental data types remain the same as in the 32-bit implementation. The 32-bit implementation is based on the ILP32 data model in which `int`, `long`, and pointers are 32-bit values. For more information about these models, see Chapter 3, “Comparing 32-Bit Interfaces and 64-Bit Interfaces”.

All new applications must be developed as 64-bit applications and many of the existing 32-bit applications might require conversion to 64-bit. You might want to convert applications with the following characteristics:

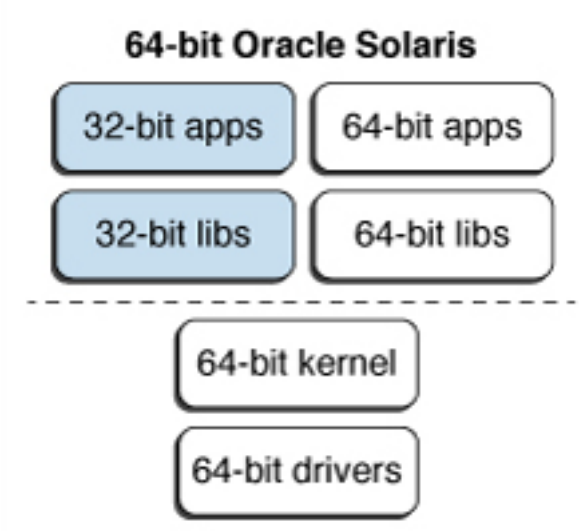
- Can benefit from more than 4 gigabytes of virtual address space
- Are restricted by 32-bit interface limitations
- Can benefit from full 64-bit registers to do efficient 64-bit arithmetic
- Can benefit from the performance improvement that the 64-bit instruction set provides
- Can benefit from the use of large file awareness
- Must run after January 2038
- Can benefit from the use of 64-bit `time_t` that can handle dates for billion years into the future
- Read and interpret kernel memory through the use of `libkvm`, `/dev/mem`, or `/dev/kmem`
- Use `/proc` to debug 64-bit processes
- Use a library that has only a 64-bit version
- Access files from a Network Attached Storage (NAS) filer that uses file inode values greater than 32-bit

2.1 Major Features of 64-Bit Development Environment

The following figure shows support for 32-bit and 64-bit applications in the Oracle Solaris 11 operating environment. The Oracle Solaris 11 OS supports 32-bit applications and libraries and

64-bit libraries and applications *simultaneously* on top of a 64-bit kernel that uses 64-bit device drivers.

FIGURE 2 The Oracle Solaris 11 Operating Environment Architecture



The major features of the 64-bit environment include support for:

- Large virtual address space
- Large files
- 64-bit arithmetic
- Removal of certain system limitations
- Dates beyond January 2038 with a 64-bit `time_t`

2.1.1 Large Virtual Address Space

In the 64-bit environment, a process can have up to 64-bit of virtual address space, that is, 18 exabytes. This virtual address space is approximately *4 billion* times the current maximum of a 32-bit process.

Note - On some systems, the full 64 bits of address space might not be available if used for other purposes via VA masking or Application Data Integrity.

2.1.2 Large Files

If an application requires only support for large files, the application can remain 32-bit and use the Large Files interface. However, if portability is not a primary concern, consider converting the application to 64-bit. A 64-bit program takes full advantage of the 64-bit capabilities with a coherent set of interfaces.

2.1.3 64-Bit Arithmetic

64-bit arithmetic has long been available in earlier 32-bit Solaris releases. However, the 64-bit implementation uses the full 64-bit system registers for integer operations and parameter passing. The 64-bit implementation allows an application to take full advantage of the capabilities of the 64-bit CPU hardware.

2.1.4 System Limitations Removed

The 64-bit system interfaces are inherently more capable than some of their 32-bit equivalents. Application programmers concerned about year 2038 problems, when 32-bit `time_t` runs out of time, can use the 64-bit `time_t`. While 2038 seems a long way off, applications that do computations for future events, such as mortgages, might require the expanded time capability.

2.2 Interoperability Issues

The interoperability issues that require an application to be made 64-bit safe or changed to interoperate with both 32-bit and 64-bit programs can include:

- Client and server transfers
- Programs that manipulate persistent data
- Shared memory

2.2.1 Kernel Memory Readers

Because the kernel is an LP64 object that uses 64-bit data structures internally, existing 32-bit applications that use `libkvm`, `/dev/mem`, or `/dev/kmem` do not work properly and must be converted to 64-bit applications.

2.2.2 `/proc` Restrictions

A 32-bit program that uses `/proc` is able to look at 32-bit processes, but cannot understand all attributes of a 64-bit process. The existing interfaces and data structures that describe the process are not large enough to contain the 64-bit values. Such programs must be recompiled as 64-bit programs in order to work with both 32-bit processes and 64-bit processes. The ability to work with both 32-bit processes and 64-bit processes is most typically a problem for debuggers.

2.2.3 64-Bit Libraries

32-bit applications are linked to 32-bit libraries, and 64-bit applications are linked to 64-bit libraries. With the exception of those libraries that have become obsolete, all of the system libraries are provided in both 32-bit versions and 64-bit versions.

2.3 Estimating the Effort of 64-Bit Conversion

After you have decided to convert your application to a full 64-bit program, it is worth noting that many applications require only a little work to accomplish that goal. The remaining chapters discuss how to evaluate your application and the effort involved in conversion.

♦ ♦ ♦ CHAPTER 3

Comparing 32-Bit Interfaces and 64-Bit Interfaces

As discussed in "1.1 Getting Past the 4 Gigabyte Barrier" on page 11, to get the benefits of a 64-bit operating system you must convert the 32-bit applications to 64-bit applications. Some applications might only need to be recompiled as 64-bit applications; others must be converted.

Note - Oracle Solaris 11 is a 64-bit only operating system.

3.1 Application Programming Interfaces

The 32-bit application programming interfaces (APIs) supported in the 64-bit operating environment are same as the APIs supported in the 32-bit operating environment. Thus, no changes are required for 32-bit applications between the 32-bit environment and 64-bit environment. However, *recompiling* as a 64-bit application can require cleanup. See the rules that are defined in Chapter 4, "Converting Applications" for guidelines on how to clean up code for 64-bit applications.

In Oracle Solaris 11 the default 64-bit APIs are basically the `UNIX 03` family of APIs. The specification of the APIs is written in terms of derived types. The 64-bit versions are obtained by expanding some of the derived types to 64-bit values. Correctly written applications that use these APIs are portable between the 32-bit environment and the 64-bit environment. For more information about supported standards in Oracle Solaris, see the `standards(5)` man page.

3.2 Application Binary Interfaces

Oracle Solaris supports the following ABI versions:

- SPARC V8 ABI – Used for 32-bit SPARC systems

- SPARC V9 ABI – Used for 64-bit SPARC systems
- i386 ABI – Used for 32-bit x86 systems
- amd64 ABI – Used for 64-bit x86 systems

For more information, see "6.1 SPARC V9 ABI Features" on page 51 and "6.2 AMD64 ABI Features" on page 55.

3.3 Compatibility Between 32-Bit Applications and 64-Bit Applications

The following sections discuss the different levels of compatibility between 32-bit applications and 64-bit applications.

3.3.1 Application Binaries

Existing 32-bit applications can run on either 32-bit or 64-bit operating environments. The only exceptions are those applications that use `libkvm`, `/dev/mem`, `/dev/kmem`, or `/proc`. For more information, see "1.1 Getting Past the 4 Gigabyte Barrier" on page 11.

3.3.2 Application Source Code

Source-level compatibility is maintained for 32-bit applications. For 64-bit applications, the principal changes are with respect to the derived types used in the APIs. Applications that use the derived types and interfaces correctly are source compatible for 32-bit, and make the transition to 64-bit easy.

3.3.3 Device Drivers

Because 32-bit device drivers cannot be used with the 64-bit operating system, these drivers must be recompiled as 64-bit objects. Moreover, the 64-bit drivers must support both 32-bit applications and 64-bit applications. All drivers supplied with the 64-bit operating environment support both 32-bit applications and 64-bit applications. However, there are no changes in the

fundamental driver model and the interfaces supported by the DDI. The principal work is to clean up the code to be correct in an LP64 environment. For more information, see *Writing Device Drivers for Oracle Solaris 11.3*.

3.4 Show That 64-Bit and 32-Bit Applications Run on Your Oracle Solaris 11 System

The Oracle Solaris 11 and later versions are 64-bit only OS. However, you can still run your 32-bit applications on the 64-bit OS. You can use the `isainfo` command to know if your system supports 32-bit and 64-bit applications.

The following is an example of the `isainfo` command executed on an UltraSPARC system running the 64-bit OS:

```
$ isainfo -v
64-bit sparcv9 applications
32-bit sparc applications
64-bit sparcv9 applications
        crc32c cbcond pause mont mpmul sha512 sha256 sha1 md5 camellia kasumi
        des aes ima hpc vis3 fmaf asi_blk_init vis2 vis popc
32-bit sparc applications
        crc32c cbcond pause mont mpmul sha512 sha256 sha1 md5 camellia kasumi
        des aes ima hpc vis3 fmaf asi_blk_init vis2 vis popc v8plus div32 mul32
```

When the same command is run on an x86 system running the 64-bit OS:

```
$ isainfo -v
64-bit amd64 applications
32-bit i386 applications
64-bit amd64 applications
        ssse3 ahf cx16 sse3 sse2 sse fxsr mmx cmov amd_sysc cx8 tsc fpu
32-bit i386 applications
        ssse3 ahf cx16 sse3 sse2 sse fxsr mmx cmov sep cx8 tsc fpu
```

Note - Not all x86 systems are capable of running the 64-bit kernel. x86 systems which are not capable of running the 64-bit kernel cannot run Oracle Solaris 11 and later versions.

One useful option of the `isainfo` command is the `-n` option, which prints the native instruction set of the running platform:

```
$ isainfo -n
sparcv9
```

The -b option prints the number of bits in the address space of the corresponding native applications environment:

```
$ echo "Welcome to "`isainfo -b`"-bit Oracle Solaris"
Welcome to 64-bit Oracle Solaris
```

For more information, see the isainfo(1) man page.

Applications that must run on earlier versions of the Oracle Solaris can ascertain whether 64-bit capabilities are available. Check the output of uname(1) or check for the existence of /usr/bin/isainfo.

Libraries that depend on instruction set extensions should use the hardware capability facility of the dynamic linker. Use the isainfo command to ascertain the instruction set extensions on the current platform. For more information, see "Generating the Output File" in *Oracle Solaris 11.3 Linkers and Libraries Guide*.

```
$ isainfo -x
amd64: sse2 sse fxsr amd_3dnowx amd_3dnow amd_mmx mmx cmov amd_sysc cx8 tsc fpu
i386: sse2 sse fxsr amd_3dnowx amd_3dnow amd_mmx mmx cmov amd_sysc cx8 tsc fpu
```

♦ ♦ ♦ CHAPTER 4

Converting Applications

This chapter provides the information that you need to convert your 32-bit applications to 64-bit applications.

When you write or modify code for both the 32-bit and 64-bit compilation environments, you face two basic issues:

- Data type consistency between the different data-type models
- Interaction between the applications using different data-type models

Maintaining a single code-source with as few `#ifdefs` as possible is usually better than maintaining multiple source trees. Therefore, this chapter provides guidelines for writing code that works correctly in both 32-bit and 64-bit compilation environments. In some cases, the conversion of current code requires only a recompilation and relinking with the 64-bit libraries. However, for those cases where code changes are required, this chapter discusses the tools and strategies that make conversion easier.

4.1 Overview of the Data Model Differences

The biggest difference between the 32-bit and the 64-bit compilation environments is the change in data-type models.

The C data-type model for 64-bit applications is the ILP32 data model, so named because `int`, `long`, and pointers are 32-bit data types. The LP64 data model, so named because `long` and pointers grow to 64-bit, is the creation of a consortium of companies across the industry. The remaining C types, `int`, `long long`, `short`, and `char`, are the same in both data-type models.

The following sample program, `foo.c`, directly illustrates the effect of the LP64 data model in contrast to the ILP32 data models. The same program can be compiled as either a 32-bit program or a 64-bit program.

```
#include <stdio.h>
int
```

```
main(int argc, char *argv[])
{
  (void) printf("char is \t\t%lu bytes\n", sizeof (char));
  (void) printf("short is \t%lu bytes\n", sizeof (short));
  (void) printf("int is \t\t%lu bytes\n", sizeof (int));
  (void) printf("long is \t\t%lu bytes\n", sizeof (long));
  (void) printf("long long is \t\t%lu bytes\n", sizeof (long long));
  (void) printf("pointer is \t%lu bytes\n", sizeof (void *));
  return (0);
}
```

The result of 32-bit compilation is:

```
$ cc -m32 -O -o foo32 foo.c
$ foo32
char is          1 bytes
short is         2 bytes
int is           4 bytes
long is          4 bytes
long long is     8 bytes
pointer is       4 bytes
```

The result of 64-bit compilation is:

```
$ cc -m64 -O -o foo64 foo.c
$ foo64
char is          1 bytes
short is         2 bytes
int is           4 bytes
long is          8 bytes
long long is     8 bytes
pointer is       8 bytes
```

Note - The default compilation mode depends on the compiler being used. To determine whether your compiler produces 64-bit or 32-bit code by default, refer to the compiler documentation.

The standard relationship between C integral types still holds true.

```
sizeof (char) <= sizeof (short) <= sizeof (int) <= sizeof (long) <= sizeof (long long)
```

Table 1, “Data Type Sizes in Bits,” on page 24 lists the basic C types and their corresponding sizes in bits in the data type models for both LP32 and LP64.

TABLE 1 Data Type Sizes in Bits

C data type	ILP32	LP64
`char`	8	8

C data type	ILP32	LP64
`short`	16	16
`int`	32	32
`long`	32	**64**
`long long`	64	64
`pointer`	32	**64**
`enum`	32	32
`float`	32	32
`double`	64	64
`long double`	128	128

Current 32-bit applications typically assume that `int`, `long`, and pointers are the same size. However, the size of `long` and pointers changes in the LP64 data model, which can cause many ILP32 to LP64 conversion problems.

In addition, declarations and casts are very important. How expressions are evaluated can be affected when the types change. The effects of standard C conversion rules are influenced by the change in data-type sizes. To adequately show what you intend, explicitly declare the types of constants. You can also use casts in expressions to make certain that the expression is evaluated the way you intend. This practice is particularly important with sign extension, where explicit casting is essential for demonstrating intent.

4.2 Implementing Single Source Code

The sections that follow describe some of the resources available to application developers that help you write single-source code that supports both 32-bit and 64-bit compilation.

The system include files `sys/types.h` and `inttypes.h` contain constants, macros, and derived types that are helpful in making applications 32-bit and 64-bit safe. While a detailed discussion of these is beyond the scope of this document, some are discussed in the sections that follow, and in Appendix A, “Changes in Derived Types”.

4.2.1 Feature Test Macros

An application source file that includes `sys/types.h` makes the definitions of the programming model symbols, `_LP64` and `_ILP32`, available when you include `sys/isa_defs.h`.

For information about preprocessor symbols (_LP64 and _ILP32) and macros (_LITTLE_ENDIAN and _BIG_ENDIAN), see the types(3HEAD) man page.

4.2.2 Derived Types

Using the system derived types to make code safe for both the 32-bit and the 64-bit compilation environment is good programming practice. When you use derived data-types, only the system derived types must change for data model changes, or porting.

The system include files sys/types.h and inttypes.h contain constants, macros, and derived types that are helpful in making applications 32-bit and 64-bit safe.

4.2.3 sys/types.h Header File

Include sys/types.h in an application source file to gain access to the definition of _LP64 and _ILP32. This header also contains a number of basic derived types that should be used whenever appropriate. In particular, the following are of special interest:

- clock_t represents the system times in clock ticks.
- dev_t is used for device numbers.
- off_t is used for file sizes and offsets.
- ptrdiff_t is the signed integral type for the result of subtracting two pointers.
- size_t reflects the size, in bytes, of objects in memory.
- ssize_t is used by functions that return a count of bytes or an error indication.
- time_t counts time in seconds.

All of these types remain 32-bit quantities in the ILP32 compilation environment and grow to 64-bit quantities in the LP64 compilation environment.

4.2.4 inttypes.h Header File

The inttypes.h include file was added to the Solaris 2.6 release to provide constants, macros, and derived types that help programmers make their code compatible with explicitly sized data items, independent of the compilation environment. It contains mechanisms for manipulating 8-

bit, 16-bit, 32-bit, and 64-bit objects. The file contains the type definitions specified in the ISO C99 standard (ISO/IEC 9899:1999) for the C programming language.

The basic features provided by `inttypes.h` are:

- Fixed-width integer types.
- Helpful types such as `uintptr_t`
- Constant macros
- Limits
- Format string macros

The following sections provide more information about the basic features of `<inttypes.h>`.

4.2.4.1 Fixed-Width Integer Types

The fixed-width integer types that `inttypes.h` provides include signed integer types such as `int8_t`, `int16_t`, `int32_t`, and `int64_t`, and unsigned integer types such as `uint8_t`, `uint16_t`, `uint32_t`, and `uint64_t`.

Derived types, defined as the smallest integer types that can hold the specified number of bits, include `int_least8_t`,..., `int_least64_t`, `uint_least8_t`,..., `uint_least64_t`.

Using an `int` or `unsigned int` for such operations as loop counters and file descriptors is safe. Using a `long` for an array index is also safe. However, do not use these fixed-width types indiscriminately. Use fixed-width types for explicit binary representations of the following items:

- On-disk data
- Over the data wire
- Hardware registers
- Binary interface specifications
- Binary data structures

4.2.4.2 Helpful Types Such as `unintptr_t`

The `inttypes.h` file includes signed and unsigned integer types large enough to hold a pointer. These are given as `intptr_t` and `uintptr_t`. In addition, `inttypes.h` provides `intmax_t` and `uintmax_t`, which are the longest (in bits) signed and unsigned integer types available.

Use the `uintptr_t` type as the integral type for pointers instead of a fundamental type such as `unsigned long`. Even though an `unsigned long` is the same size as a pointer in both the ILP32 and LP64 data models, using `uintptr_t` means that only the definition of `uintptr_t` is affected if the data model changes. This method makes your code portable to many other systems and is also a clearer way to express your intentions in C.

The `intptr_t` and `uintptr_t` types are extremely useful for casting pointers when you want to perform address arithmetic. Use `intptr_t` and `uintptr_t` types instead of `long` or unsigned `long` for this purpose.

Note - Use of `uintptr_t` for casting is usually safer than `intptr_t`, especially for comparisons.

4.2.4.3 Constant Macros

Macros are provided to specify the size and sign of a given constant. The macros are `INT8_C(c)`, ..., `INT64_C(c)`, `UINT8_C(c)`,..., `UINT64_C(c)`. Basically, these macros place an `l`, `ul`, `ll`, or `ull` at the end of the constant, if necessary. For example, `INT64_C(2)` appends `ll` to the constant `2` for ILP32 and an `l` for LP64.

Macros for making a constant the biggest type are `INTMAX_C(c)` and `UINTMAX_C(c)`. These macros can be very useful for specifying the type of constants described in “4.4 Guidelines for Converting to LP64 Data Type Model” on page 31.

4.2.4.4 Limits in `inttypes.h`

The limits defined by `inttypes.h` are constants specifying the minimum and maximum values of various integer types. This includes minimum and maximum values of each of the fixed-width types, such as `INT8_MIN`,..., `INT64_MIN`, `INT8_MAX`,..., `INT64_MAX`, and their unsigned counterparts.

The minimum and maximum for each of the least-sized types are given, too. These include `INT_LEAST8_MIN`,..., `INT_LEAST64_MIN`, `INT_LEAST8_MAX`,..., `INT_LEAST64_MAX`, and their unsigned counterparts.

Finally, the minimum and maximum value of the largest supported integer types is defined. These include `INTMAX_MIN` and `INTMAX_MAX` and their corresponding unsigned versions.

For more information, see the `inttypes.h`(3HEAD) man page.

4.2.4.5 Format String Macros

The inttypes.h file includes the macros that specify the printf() and scanf() format specifiers. Essentially, these macros prepend the format specifier with an l or ll to identify the argument as a long or long long, given that the number of bits in the argument is built into the name of the macro.

Some macros for printf(3C) print both the smallest and largest integer types in decimal, octal, unsigned, and hexadecimal formats, as shown in the following example.

```
int64_t i;
printf("i =%" PRIx64 "\n", i);
```

Similarly, macros for scanf(3C) read both the smallest and largest integer types in decimal, octal, unsigned, and hexadecimal formats.

```
uint64_t u;
scanf("%" SCNu64 "\n", &u);
```

Do not use these macros indiscriminately. They are best used in conjunction with the fixed-width types. For more information, see “4.2.4.1 Fixed-Width Integer Types” on page 27.

4.3 Finding Errors With lint

The lint program from Oracle Developer Studio allows you to detect issues in the code such as non-portable codes, unreachable statements. The -errchk option detects potential 64-bit porting problems. You can also specify cc -v, which directs the compiler to perform additional and stricter semantic checks. The -v option also enables certain lint-like checks on the named files.

When you enhance code to be 64-bit safe, use the header files present in the Oracle Solaris operating system because these files have the correct definition of the derived types and data structures for the 64-bit compilation environment.

Use lint to check code that is written for both the 32-bit and the 64-bit compilation environment. Specify the -errchk=longptr64 option to generate LP64 warnings. Also use the -errchk=longptr64 flag, which checks portability to an environment for which the size of long integers and pointers is 64-bit and the size of plain integers is 32-bit. The -errchk=longptr64 flag checks assignments of pointer expressions and long integer expressions to plain integers, even when explicit casts are used.

Use the `-errchk=longptr64,signext,sizematch` option to find code where the normal ISO C value-preserving rules allow the extension of the sign of a signed-integral value in an expression of unsigned-integral type.

Use the `-m64` option of `lint` when you want to check code that you intend to run in the Oracle Solaris 64-bit compilation environment only.

For a description of `lint` options, see the Oracle Developer Studio 12.6: C User's Guide.

`lint` warnings show the line number of the offending code, a message that describes the problem, and an indication of whether a pointer is involved. The warning message also indicates the sizes of the involved data types. When you know a pointer is involved and you know the size of the data types, you can find specific 64-bit problems and avoid the preexisting problems between 32-bit and smaller types.

Be aware, however, that even though `lint` gives warnings about potential 64-bit problems, it cannot detect all problems. Also, in many cases, code that is intentional and correct for the application generates a warning.

You can suppress the warning for a given line of code by placing a comment of the form `"NOTE(LINTED("<optional message">))"` on the previous line. This comment directive is useful when you want `lint` to ignore certain lines of code such as casts and assignments. When you use `NOTE`, include `<note.h>`. For more information, see the lint-1 man page.

Caution - Exercise extreme care when you use the `"NOTE(LINTED("<optional message">))"` comment because it can mask real problems.

The sample program and `lint` output following illustrate most of the `lint` warnings that can arise in code that is not 64-bit clean.

```
 1 #include <inttypes.h>
 2 #include <stdio.h>
 3 #include <stdlib.h>
 4
 5 static char chararray[] = "abcdefghijklmnopqrstuvwxyz";
 6
 7 static char *myfunc(int i)
 8 {
 9  return(& chararray[i]);
10 }
11
12 void main(void)
13 {
14  int intx;
15  long longx;
```

```
  16  char *ptrx;
  17
  18  (void) scanf("%d", &longx);
  19  intx = longx;
  20  ptrx = myfunc(longx);
  21  (void) printf("%d\n", longx);
  22  intx = ptrx;
  23  ptrx = intx;
  24  intx = (int)longx;
  25  ptrx = (char *)intx;
  26  intx = 2147483648L;
  27  intx = (int) 2147483648L;
  28  ptrx = myfunc(2147483648L);
  29 }

(19) warning: assignment of 64-bit integer to 32-bit integer
(20) warning: passing 64-bit integer arg, expecting 32-bit integer: myfunc(arg 1)
(22) warning: improper pointer/integer combination: op "="
(22) warning: conversion of pointer loses bits
(23) warning: improper pointer/integer combination: op "="
(23) warning: cast to pointer from 32-bit integer
(24) warning: cast from 64-bit integer to 32-bit integer
(25) warning: cast to pointer from 32-bit integer
(26) warning: 64-bit constant truncated to 32 bits by assignment
(27) warning: cast from 64-bit integer constant expression to 32-bit integer
(28) warning: passing 64-bit integer constant arg, expecting 32-bit integer: myfunc(arg
 1)
function argument ( number ) type inconsistent with format
    scanf (arg 2)  long * :: (format) int * t.c(18)
    printf (arg 2)  long  :: (format) int  t.c(21)
```

(The `lint` warning that arises from line 27 of this code sample is issued when the constant expression dpes not fit into the type into which it is being cast.)

Warnings for a given source line can be suppressed by placing a `/*LINTED*/` comment on the previous line. This is useful where you have really intended the code to be a specific way. However, the Oracle Developer Studio NOTE mechanism allows more fine-grained control. An example might be in the case of casts and assignments. For more information, see lint Directives.

4.4 Guidelines for Converting to LP64 Data Type Model

When using `lint`, remember that not all problems result in `lint` warnings, nor do all `lint` warnings indicate that a change is required. Examine each possibility for intent. The examples

that follow illustrate some of the more common problems you are likely to encounter when converting code. Where appropriate, the corresponding `lint` warnings are shown.

4.4.1 Do Not Assume `int` and Pointers Are the Same Size

Because integers and pointers are the same size in the ILP32 compilation environment, some code relies on this assumption. Pointers are often cast to `int` or `unsigned int` for address arithmetic. Instead, cast your pointers to `long` because `long` and pointers are the same size in both ILP32 and LP64 data-type models. Rather than explicitly using `unsigned long`, use `uintptr_t` instead. It expresses your intent more closely and makes the code more portable, insulating it against future changes. Consider the following example:

```
char *p;
p = (char *) ((int)p & PAGEOFFSET);
%
warning: conversion of pointer loses bits
```

The modified version is:

```
char *p;
p = (char *) ((uintptr_t)p & PAGEOFFSET);
```

4.4.2 Do Not Assume `int` and `long` Are the Same Size

Because `int` and `long` were never really distinguished in ILP32, a lot of existing code uses them indiscriminately while implicitly or explicitly assuming that they are interchangeable. Any code that makes this assumption must be changed to work for both ILP32 and LP64. While an `int` and a `long` are both 32-bit in the ILP32 data model, in the LP64 data model, a `long` is 64-bit. For example:

```
int waiting;
long w_io;
long w_swap;
...
waiting = w_io + w_swap;

%
warning: assignment of 64-bit integer to 32-bit integer
```

Furthermore, large arrays of `long` or `unsigned long` can cause serious performance degradation in the LP64 data-type model as compared to arrays of `int` or `unsigned int`. Large arrays of `long` or `unsigned long` can also cause significantly more cache misses and consume more memory.

Therefore, if `int` works just as well as long for the application purposes, use `int` rather than `long`.

This argument also applies to using arrays of `int` instead of arrays of pointers. Some C applications suffer from serious performance degradation after conversion to the LP64 data-type model because they rely on many large arrays of pointers.

For more information about the capabilities of the C compilers and `lint`, see Oracle Developer Studio 12.6: C User's Guide.

4.4.3 Sign Extension

Unintended sign extension is a common problem when converting to 64-bit. It is hard to detect this problem before it occurs. However, `lint` command warns about sign-extension with `-errchk=signext`. Furthermore, the type conversion and promotion rules are somewhat obscure. To fix unintended sign extension problems, you must use explicit casting to achieve the intended results.

To understand why sign extension occurs, it helps to understand the conversion rules for ISO C. The conversion rules that seem to cause the most sign extension problems between the 32-bit and the 64-bit compilation environment come into effect during the following operations:

1. Integral promotion

   You can use a `char`, `short`, `enumerated type`, or bit-field, whether signed or unsigned, in any expression that calls for an integer.

   If an integer can hold all possible values of the original type, the value is converted to an integer; otherwise, the value is converted to an unsigned integer.
2. Conversion between signed and unsigned integers

   When an integer with a negative sign is promoted to an unsigned integer of the same or larger type, it is first promoted to the signed equivalent of the larger type and then converted to the unsigned value.

For more information about the conversion rules, refer to the ISO C standard. Also included in this standard are useful rules for ordinary arithmetic conversions and integer constants.

When the following example is compiled as a 64-bit program, the `addr` variable becomes sign-extended, even though both `addr` and `a.base` are unsigned types.

EXAMPLE 1 `test.c` Example

```
$ cat test.c
struct foo {
unsigned int base:19, rehash:13;
};

main(int argc, char *argv[])
{
  struct foo a;
  unsigned long addr;

  a.base = 0x40000;
  addr = a.base << 13;  /* Sign extension here! */
  printf("addr 0x%lx\n", addr);

 addr = (unsigned int)(a.base << 13); /* No sign extension here! */
 printf("addr 0x%lx\n", addr);
}
```

This sign extension occurs because the conversion rules are applied as follows:

- `a.base` is converted from an unsigned `int` to an `int` because of the integral promotion rule. Thus, the expression `a.base << 13` is of type `int`, but no sign extension has yet occurred.
- The expression `a.base << 13` is of type `int`, but it is converted to a `long` and then to an unsigned `long` before being assigned to `addr`, because of signed and unsigned integer promotion rules. The sign extension occurs when it is converted from an `int` to a `long`.

```
$ cc -o test64 -m64 test.c
$ ./test64
addr 0xffffffff80000000
addr 0x80000000
```

When this same example is compiled as a 32-bit program it does not display any sign extension:

```
$ cc -o test -m32 test.c
$ test

addr 0x80000000
addr 0x80000000
```

For more information about the conversion rules, refer to the ISO C standard. Also included in this standard are useful rules for ordinary arithmetic conversions and integer constants.

4.4.4 Use Pointer Arithmetic Instead of Integers

Using pointer arithmetic usually works better than integers because pointer arithmetic is independent of the data model, whereas integers might not be. Also, you can usually simplify your code by using pointer arithmetic. Consider the following example:

```
int *end;
int *p;
p = malloc(4 * NUM_ELEMENTS);
end = (int *)((unsigned int)p + 4 * NUM_ELEMENTS);

%
warning: conversion of pointer loses bits
```

The modified version is:

```
int *end;
int *p;
p = malloc(sizeof (*p) * NUM_ELEMENTS);
end = p + NUM_ELEMENTS;
```

4.4.5 Internal Data Structures

Check the internal data structures in an application for holes. Use extra padding between fields in the structure to meet alignment requirements. This extra padding is allocated when `long` or pointer fields grow to 64-bit for the LP64 data-type model. In the 64-bit compilation environment on SPARC platforms, all types of structures are aligned to the size of the largest member within them. When you repack a structure, follow the simple rule of moving the `long` and pointer fields to the beginning of the structure. However, this rule for repacking might affect performance depending on which members end up on the same cache line. Consider the following structure definition:

```
struct bar {
  int i;
  long j;
  int k;
  char *p;
};   /* sizeof (struct bar) = 32 */
```

The following example shows the same structure with the `long` and pointer data types defined at the beginning of the structure:

```
struct bar {
```

```
  char *p;
  long j;
  int i;
  int k;
};   /* sizeof (struct bar) = 24 */
```

Note - The alignment of fundamental types are different in the i386 and amd64 ABIs. See “6.3 Alignment Issues” on page 57.

4.4.6 Check Unions

Be sure to check unions because their fields might have changed sizes between ILP32 and LP64. For example:

```
typedef union {
       double   _d;
       long _l[2];
} llx_t;
```

The modified version is:

```
typedef union {
       double _d;
       int _l[2];
} llx_t;
```

4.4.7 Specify Constant Types

A lack of precision can cause the loss of data in some constant expressions. Be explicit when you specify the data types in your constant expression. Specify the type of each integer constant by adding some combination of {u,U,l,L}. You can also use casts to specify the type of a constant expression. Consider the following example:

```
int i = 32;
long j = 1 << i;  /* j will get 0 because RHS is integer expression */
```

The modified version is:

```
int i = 32;
long j = 1L << i;
```

4.4.8 Beware of Implicit Declaration

If you use `--std=c90` or `--xc99=none`, the C compiler assumes that any function or variable that is used in a module and is not defined or declared externally is an integer. Any `long` and pointer data used in this way is truncated by the compiler's implicit integer declaration. Place the appropriate `extern` declaration for the function or variable in a header and not in the C module. Include this header in any C module that uses the function or variable. Even if the function or variable is defined by the system headers, you must include the proper header in the code. Consider the following example:

```
int
main(int argc, char *argv[])
{
  char *name = getlogin()
  printf("login = %s\n", name);
  return (0);
}

%
warning: improper pointer/integer combination: op "="
warning: cast to pointer from 32-bit integer
implicitly declared to return int
getlogin        printf
```

The proper headers are now in the following modified version:

```
#include <unistd.h>
#include <stdio.h>

int
main(int argc, char *argv[])
{
  char *name = getlogin();
  (void) printf("login = %s\n", name);
  return (0);
}
```

4.4.9 `sizeof unsigned long` in LP64

In the LP64 data-type model, `sizeof()` has the effective type of an `unsigned long`. Occasionally, `sizeof()` is passed to a function expecting an argument of type `int`, or assigned or cast to an integer. In some cases, this truncation causes loss of data.

```
long a[50];
unsigned char size = sizeof (a);

%
warning: 64-bit constant truncated to 8 bits by assignment
warning: initializer does not fit or is out of range: 0x190
```

4.4.10 Use Casts to Show Your Intentions

Relational expressions can be tricky because of conversion rules. You should be very explicit about how you want the expression to be evaluated by adding casts wherever necessary.

4.4.11 Check Format String Conversion Operation

The format strings for `printf`(3C), `sprintf`(3C), `scanf`(3C), and `sscanf`(3C) might need to be changed for `long` or pointer arguments. For pointer arguments, the conversion operation given in the format string should be `%p` to work in both the 32-bit and 64-bit environments. For example:

```
char *buf;
struct dev_info *devi;
...
(void) sprintf(buf, "di%x", (void *)devi);

%
warning: function argument (number) type inconsistent with format
sprintf (arg 3)     void *: (format) int
```

The modified version is:

```
char *buf;
struct dev_info *devi;
...
(void) sprintf(buf, "di%p", (void *)devi);
```

Also check to be sure that the storage pointed to by `buf` is large enough to contain 16 digits. For `long` arguments, the `long` size specification, `l`, should be prepended to the conversion operation character in the format string. For example:

```
    size_t nbytes;
```

```
    ulong_t align, addr, raddr, alloc;
    printf("kalloca:%d%%%d from heap got %x.%x returns %x\n",
            nbytes, align, (int)raddr, (int)(raddr + alloc), (int)addr);
```

produces the warnings:

```
warning: cast of 64-bit integer to 32-bit integer
warning: cast of 64-bit integer to 32-bit integer
warning: cast of 64-bit integer to 32-bit integer
```

The following code will produce clean results:

```
    size_t nbytes;
    ulong_t align, addr, raddr, alloc;
    printf("kalloca:%lu%%%lu from heap got %lx.%lx returns %lx\n",
            nbytes, align, raddr, raddr + alloc, addr);
```

Note - The PRI* macros available in `inttypes.h` can help make format strings portable between 32-bit and 64-bit environment.

4.4.12 Compiling LP64 Programs

The following guidelines can help you increase the performance when converting to 64-bit applications:

- When compiling applications in 64-bit, it supports largefile by default. Therefore, the following `CPPFLAGS` are no longer required when compiling:
  - `CPPFLAGS += -D_FILE_OFFSET_BITS=64`
  - `CPPFLAGS += -D_LARGEFILE64_SOURCE`
  - `CPPFLAGS += -D_LARGEFILE_SOURCE`
- For single-threaded code, use `putc_unlocked()` function instead of `putc()` function and use `getc_unlocked()` function instead of `getc()` function. You can also use buffers.
- When converting the code to 64-bit, replace the explicit 64-bit interfaces and data types with just the generic interfaces and data types. For example, no need to use `fopen64()` or `off64_t()`.
- When converting code to 64-bit, ensure that all the `dlopen()` calls in the code are calling a 64-bit library.
- When compiling an LP64 program pay attention to variables that are declared `long`. There might be problems in the code in assuming an `int` and `long` are the same. For example, when assigning a value of a `long` variable to an `int` variable.

- In code that uses the `select()` interfaces, especially the `fd` masks, check to see how big a set of `fds` you actually want to handle. The default value for `FD_SETSIZE` is 1024 in 32-bit and 65536 in 64-bit. If they are careful, programs can `#define FD_SETSIZE` to another value before including any system headers to choose a different size. If you do not override it, all `fd_mask` variables will grow to be 8k (often on the stack), and operations such as `FD_ZERO`, copying `fd_masks`, or searching for the set bits in a mask returned by `select()` function will have to operate on 8k at a time.
- Without the 256 `fd` limit of the 32-bit `stdio`, 64-bit programs will run with higher `fd` limits, so it becomes even more important to use `closefrom()` or `fdwalk()` instead of `for (i = 0; i < max_fd ; i++)` loops. For more information, see `closefrom`(3C) and `fdwalk`(3C) man pages.

4.5 Other Considerations When Converting to 64-bit

The remaining guidelines highlight common problems encountered when converting an application to a full 64-bit program.

4.5.1 Check for Derived Types That Have Grown in Size

A number of derived types now represent 64-bit quantities in the 64-bit application compilation environment. This change does not affect 32-bit applications; however, any 64-bit applications that consume or export data described by these types must be re-evaluated. For example, in applications that directly manipulate the `utmp` or `utmpx` files, do not attempt to directly access these files. Instead, for correct operation in the 64-bit application environment, use the `getutxent()` and related family of functions. For more information, see `getutxent`(3C), `utmpx`(4), and `utmp`(4) man pages.

A list of changed derived types is included in Appendix A, “Changes in Derived Types”.

4.5.2 Check for Side Effects of Changes

Be aware that a type change in one area can result in an unexpected 64-bit conversion in another area. For example, check all the callers of a function that previously returned an `int` and now returns an `ssize_t`.

4.5.3 Check Whether Literal Uses of `long` Still Make Sense

A variable that is defined as a `long` is 32-bit in the ILP32 data-type model and 64-bit in the LP64 data-type model. Where possible, avoid problems by redefining the variable and use a more portable derived type.

Related to this issue, a number of derived types have changed under the LP64 data-type model. For example, `pid_t` remains a `long` in the 32-bit environment, but under the 64-bit environment, a `pid_t` is an `int`. For a list of derived types modified for the LP64 compilation environment, see Appendix A, “Changes in Derived Types”.

4.5.4 Check Use `#ifdef` for Explicit 32-Bit Versus 64-Bit Prototypes

In some cases, specific 32-bit and 64-bit versions of an interface are unavoidable. You can distinguish these versions by specifying the `_LP64` or `_ILP32` feature test macros in the headers. Similarly, code that runs in 32-bit and 64-bit environments needs to use the appropriate `#ifdefs`, depending on the compilation mode.

4.5.5 Calling Convention Changes

When you pass structures by value and compile the code for a 64-bit environment, the structure is passed in registers rather than as a pointer to a copy if it is small enough. This process can cause problems if you try to pass structures between C code and handwritten assembly code.

Floating-point parameters work in a similar fashion: some floating-point values passed by value are passed in floating-point registers.

4.5.6 Algorithmic Changes

After code has been made 64-bit safe, review it again to verify that the algorithms and data structures still make sense. The data types are larger, so data structures might use more space.

The performance of your code might change as well. Given these concerns, you might need to adapt your code appropriately.

4.6 Checklist for Converting to 64-bit

The following checklist might be helpful to convert your code to 64-bit:

- Read this entire document with an emphasis on the "4.4 Guidelines for Converting to LP64 Data Type Model" on page 31.
- Review all data structures and interfaces to verify that these are still valid in the 64-bit environment.
- Include `<sys/types.h>` in your code to pull in the `_ILP32` or `_LP64` definitions as well as many basic derived types.
- Move function prototypes and external declarations with non-local scope to headers and include these headers in your code.
- Run `lint` using `-errchk=longptr64` and review each warning individually, being aware that not all warnings require a change to the code. Depending on the resulting changes, you might also want to run `lint` again, both in 32-bit and 64-bit modes.
- Compile code as both 32-bit and 64-bit, unless the application is being provided only as 64-bit.
- Test the application by executing the 32-bit and the 64-bit versions on the 64-bit operating system. The output of a 32-bit binary should be the same as the output of the 64-bit binary.

4.7 Sample 64-Bit From 32-Bit Program

The following sample program, `foo.c`, directly illustrates the effect of the LP64 data model in contrast to the ILP32 data models. The same program can be compiled as either a 32-bit program or a 64-bit program.

```
#include <stdio.h>
int
main(int argc, char *argv[])
{
  (void) printf("char is \t\t%lu bytes\n", sizeof (char));
  (void) printf("short is \t%lu bytes\n", sizeof (short));
  (void) printf("int is \t\t%lu bytes\n", sizeof (int));
  (void) printf("long is \t\t%lu bytes\n", sizeof (long));
  (void) printf("long long is \t\t%lu bytes\n", sizeof (long long));
```

```
  (void) printf("pointer is \t%lu bytes\n", sizeof (void *));
  return (0);
}
```

The result of 32-bit compilation is:

```
$ cc -m32 -O -o foo32 foo.c
$ foo32
char is          1 bytes
short is         2 bytes
int is           4 bytes
long is          4 bytes
long long is     8 bytes
pointer is       4 bytes
```

The result of 64-bit compilation is:

```
$ cc -m64 -O -o foo64 foo.c
$ foo64
char is          1 bytes
short is         2 bytes
int is           4 bytes
long is          8 bytes
long long is     8 bytes
pointer is       8 bytes
```

Note - The default compilation mode depends on the compiler being used. To determine whether your compiler produces 64-bit or 32-bit code by default, refer to the compiler documentation.

♦ ♦ ♦ CHAPTER 5

64-Bit Development Environment

This chapter explains the 64-bit application development environment. It also describes the build environment, including header and library issues, compiler options, linking, debugging tools and provides guidance on packaging issues.

If you have come this far, the assumption is that you are using a 64-bit version. To confirm this, you can use the `isainfo(1)` command that was explained in Chapter 3, "Comparing 32-Bit Interfaces and 64-Bit Interfaces".

5.1 64-Bit Build Environment

The build environment includes the system headers, compilation system, and libraries. These are explained in the following sections.

5.1.1 64-Bit Header Files

A single set of system headers supports both 32-bit and 64-bit compilation environments. You need not specify a different include path for the 64-bit compilation environment. Starting with Oracle Solaris 11, the <`system/header`> is available as an IPS package that contains core C and C++ header files.

To understand the changes made to the headers to support the 64-bit environment, you should understand the various definitions in the <`sys/isa_defs.h`> header file. This header contains a group of well known `#defines` and sets these for each instruction set architecture. Inclusion of <`sys/types.h`> automatically includes <`sys/isa_defs.h`>.

The symbols in the following table are defined by the compilation environment:

Symbol	Description
`__sparc`	Indicates any of the SPARC family of processor architectures. This includes SPARC V7, SPARC V8, and SPARC V9 architectures. The symbol `sparc` is a deprecated historical synonym for `__sparc`.
`__sparcv8`	Indicates the 32-bit SPARC V8 architecture as defined by Version 8 of the *SPARC Architecture Manual*.
`__sparcv9`	Indicates the 64-bit SPARC V9 architecture as defined by Version 9 of the *SPARC Architecture Manual*.
`__x86`	Indicates any of the x86 family of processor architectures.
`__i386`	Indicates the 32-bit i386 architecture.
`__amd64`	Indicates the 64-bit amd64 architecture.

Note - `__i386` and `__amd64` are mutually exclusive. The `__sparcv8` and `__sparcv9` symbols are mutually exclusive and are only relevant when the `__sparc` symbol is defined.

The following symbols are derived from a combination of the symbols defined above:

`_ILP32` The data model where sizes of `int`, `long`, and pointer are all 32-bit.

`_LP64` The data model where sizes of `long` and pointer are all 64-bit.

Note - The `_ILP32` and `_LP64` symbols are mutually exclusive.

If writing completely portable code is not possible, and specific 32-bit versus 64-bit code is required, make the code conditional by using `_ILP32` or `_LP64`. This makes the compilation environment system independent and maximizes the portability of the application to all 64-bit platforms.

5.1.2 32-Bit and 64-Bit Libraries

The Oracle Solaris operating environment provides shared libraries for both 32-bit and 64-bit compilation environments.

32-bit applications must link with 32-bit libraries, and 64-bit applications must link with 64-bit libraries. It is not possible to create or execute a 32-bit application by using 64-bit libraries. The 32-bit libraries continue to be located in `/usr/lib` and `/lib`. The 64-bit libraries are located in a subdirectory of the appropriate `lib` directory. Because the placement of the 32-bit libraries has not changed, 32-bit applications built on prior releases are binary compatible. Portable Makefiles should refer to any library directories by using the `64` symbolic links. The `/usr/`

`lib/64` is symbolic link to `/usr/lib/sparcv9` on SPARC systems and `/usr/lib/amd64` on x86 systems.

In order to build 64-bit applications, you need 64-bit libraries. The compiler and other tools such as `ld`, `ar`, and `as` are capable of building 64-bit programs and 32-bit programs. Of course, a 64-bit program built on a system running the 32-bit operating system cannot execute in that 32-bit environment. However, in the future releases of Oracle Solaris, 32-bit might not be the default compilation mode.

5.2 Linking Object Files

Use the `ld` link-editor to link to object files. `ld` is a cross link-editor that can link 32-bit objects or 64-bit objects for SPARC or x86 systems. `ld` uses the ELF class on system type of the first relocatable object on the command line to govern the mode in which to operate. If the linker is presented with a collection of ELF32 object files as input, it creates an ELF32 output file; similarly, if it is presented with a collection of ELF64 object files as input, it creates an ELF64 output file. Attempts to mix ELF32 and ELF64 input files are rejected by the linker.

5.2.1 `LD_LIBRARY_PATH` Environment Variable

The following table lists the dynamic linkers for SPARC and x86 platforms:

TABLE 2 Dynamic Linkers

System Architecture	Linker For 32-bit Applications	Linker For 64-bit Applications
SPARC	`/usr/lib/ld.so.1`	`/usr/lib/sparcv9/ld.so.1`
x86	`/usr/lib/ld.so.1`	`/usr/lib/amd64/ld.so.1`

For more information, see the `ld(1)` man page.

At runtime, both dynamic linkers search the *same* list of colon-separated directories specified by the `LD_LIBRARY_PATH` environment variable. However, the 32-bit dynamic linker binds only to 32-bit libraries, while the 64-bit dynamic linker binds only to 64-bit libraries. So directories containing both 32-bit and 64-bit libraries can be specified via `LD_LIBRARY_PATH`, if needed.

The 64-bit dynamic linker's search path can be overridden by using the `LD_LIBRARY_PATH_64` environment variable and 32-bit dynamic linker search path can be overridden by using the `LD_LIBRARY_PATH_32` environment variable.

5.2.2 `$ORIGIN` Keyword

A common technique for distributing and managing applications is to place related applications and libraries in a directory hierarchy. Typically, the libraries used by the applications reside in a `lib` subdirectory, while the applications themselves reside in a `bin` subdirectory of a base directory. This base directory can then be exported by using NFS and mounted on client systems. In some environments, the automounter and the name service can be used to distribute the applications, and to ensure the file system namespace of the application hierarchy is the same on all clients. In such environments, the applications can be built by using the `-R` flag to the linker to specify the absolute path names of the directories that should be searched for shared libraries at runtime.

However, in other environments, the file system namespace is not so well controlled, and developers have resorted to using a debugging tool – the `LD_LIBRARY_PATH` environment variable – to specify the library search path in a wrapper script. This is unnecessary, because the `$ORIGIN` keyword can be used in path names specified to the linker `-R` option. The `$ORIGIN` keyword is expanded at runtime to be the name of the directory where the executable itself is located. This effectively means that the path name to the library directory can be specified by using the pathname relative to `$ORIGIN`. This allows the application base directory to be relocated without having to set `LD_LIBRARY_PATH` at all.

This functionality is available for both 32-bit and 64-bit applications, and it is well worth considering when creating new applications to reduce the dependencies on users or scripts correctly configuring `LD_LIBRARY_PATH`.

For more information, see *Oracle Solaris 11.3 Linkers and Libraries Guide*.

5.3 Packaging 32-Bit and 64-Bit Applications

The following sections discuss packaging considerations for 32-bit and 64-bit applications.

5.3.1 Placement of Libraries and Programs

The placement of new libraries and programs follows the standard conventions described in “5.1.2 32-Bit and 64-Bit Libraries” on page 46. The 32-bit libraries continue to be located in the same place, while the 64-bit libraries should be placed in the specific architecture-

dependent directory under the normal default directories. Placement of 32-bit and 64-bit specific applications should be transparent to the user.

For SPARC systems, the 32-bit libraries should be placed in the same library directories. And the 64-bit libraries should be placed in the `sparcv9` subdirectory under the appropriate `lib` directory. Programs that require versions specific to 32-bit or 64-bit environments are a slightly different case. These should be placed in the appropriate `sparcv7` or `sparcv9` subdirectory of the directory where they are normally located.

For x86 and AMD systems, the 64-bit libraries should be placed in the amd64 subdirectory under the appropriate `lib` directory. Programs that require versions specific to 32-bit or 64-bit environments should be placed in the appropriate `i86` or `amd64` subdirectory of the directory where they are normally located.

For more information, see “5.3.3 Application Naming Conventions” on page 49.

5.3.2 Packaging Guidelines

Packaging options include creating specific packages for 32-bit and 64-bit applications, or combining the 32-bit and 64-bit versions in a single package. In the case where a single package is created, you should use the subdirectory naming convention for the contents of the package, as described in this chapter.

5.3.3 Application Naming Conventions

Rather than having specific names for 32-bit and 64-bit versions of an application, such as `foo32` and `foo64`, 32-bit and 64-bit applications can be placed in the appropriate platform-specific subdirectory, as explained in “5.3.1 Placement of Libraries and Programs” on page 48. Wrappers, which are explained in the next section, can then be used to run the correct version of the application. One advantage is that the user does not need to know about the specific 32-bit and 64-bit version, since the correct version executes automatically, depending on the platform.

5.4 Debugging 64-Bit Applications

All of the Oracle Solaris debugging tools have been updated to work with 64-bit applications. This includes the `truss(1)` command, the `/proc` tools (`proc(1)`) and `mdb`.

The `dbx` debugger, capable of debugging 64-bit applications, is available as part of the Oracle Developer Studio tool suites. The remaining tools are included with the Oracle Solaris release.

To debug programs that use the `amd64` ABI, additional flags such as `-preserve_argvalues` should be passed to the Oracle Developer Studio compiler. This allows you to save registers on the stack.

A number of enhancements are available in `mdb` for debugging 64-bit programs. As expected, using "*" to dereference a pointer will dereference 8 bytes for 64-bit programs and 4 bytes for 32-bit programs. In addition, the following modifiers are available:

```
Additional ?, /, = modifiers:

g  (8) Display 8 bytes in unsigned octal
G  (8) Display 8 bytes in signed octal
e  (8) Display 8 bytes in signed decimal
E  (8) Display 8 bytes in unsigned decimal
J  (8) Display 8 bytes in hexadecimal
K  (n) Print pointer or long in hexadecimal
    Display 4 bytes for 32-bit programs
    and 8 bytes for 64-bit programs.
y  (8) Print 8 bytes in date format

Additional ? and / modifiers:

M <value> <mask>  Apply <mask> and compare for 8-byte value;
    move '.' to matching location.
Z  (8) write 8 bytes
```

♦ ♦ ♦ CHAPTER 6

Advanced Topics

This chapter presents a collection of miscellaneous programming topics for systems programmers who want to understand more about the 64-bit Oracle Solaris operating environment.

Most of the new features of the 64-bit environment are extensions of generic 32-bit interfaces, though several new features are unique to 32-bit environments.

6.1 SPARC V9 ABI Features

64-bit applications are described by using Executable and Linking Format (ELF64), which allows large applications and large address spaces to be described completely.

SPARCV9. The *SPARC Compliance Definition*, Version 2.4.1, contains details of the SPARC V9 ABI. It describes the 32-bit SPARC V8 ABI and the 64-bit SPARC V9 ABI.

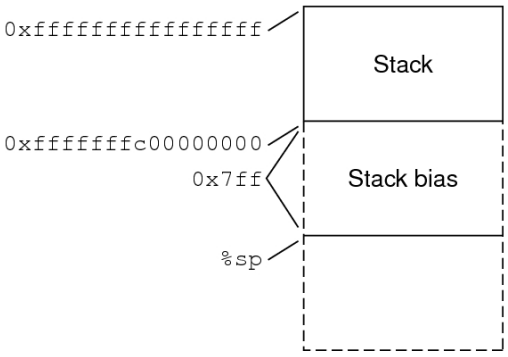
Following is a list of the SPARC V9 ABI features.

- The SPARC V9 ABI allows all 64-bit SPARC instructions and 64-bit wide registers to be used to their full effect. Many of the new relevant instructions are extensions of the existing V8 instruction set. For more information, see Oracle SPARC Servers.
- The basic calling convention is the same. The first six arguments of the caller are placed in the out registers `%o0-%o5`. The SPARC V9 ABI still uses a register window on a larger register file to make calling a function a "cheap" operation. Integer and pointer results are returned in `%o0` and floating-point results are returned differently on both 32-bit and 64-bit ABIs. Because all registers are now treated as 64-bit, 64-bit values can now be passed in a single register, rather than a register pair.
- The layout of the stack is different and the basic cell size is increased from 32-bit to 64-bit. The stack is always aligned on a 16-byte boundary for a 64-bit code and memory allocators like `malloc()` are expected to return 16-byte aligned data. The return address is still `%o7 + 8`.

- `%o6` is still referred to as the *stack pointer* register `%sp`, and `%i6` is the *frame pointer* register `%fp`. However, the `%sp` and `%fp` registers are offset by a constant, known as the *stack bias*, from the actual memory location of the stack. The size of the stack bias is 2047 bytes.
- Instruction sizes are still 32-bit. Address constant generation therefore takes more instructions. The call instruction can no longer be used to branch anywhere in the address space, since it can only reach within plus or minus 2 gigabytes of `%pc`.
- Integer multiply and divide functions are now implemented completely in hardware.
- Structure passing and return are accomplished differently. Small data structures and some floating point arguments are now passed directly in registers.
- User traps allow certain traps from non-privileged code to be handled by a user trap handler (instead of delivering a signal).
- Most of the data types are now aligned to their size.
- Many basic derived types are larger. Thus many system call interface data structures are now of different sizes.
- Two different sets of libraries exist on the system: those for 32-bit SPARC applications and those for 64-bit SPARC applications.

6.1.1 Stack Bias

An important feature of the SPARC V9 ABI for developers is the stack bias. For 64-bit SPARC programs, a stack bias of 2047 bytes must be added to both the frame pointer and the stack pointer to get to the actual data of the stack frame. See the following figure.

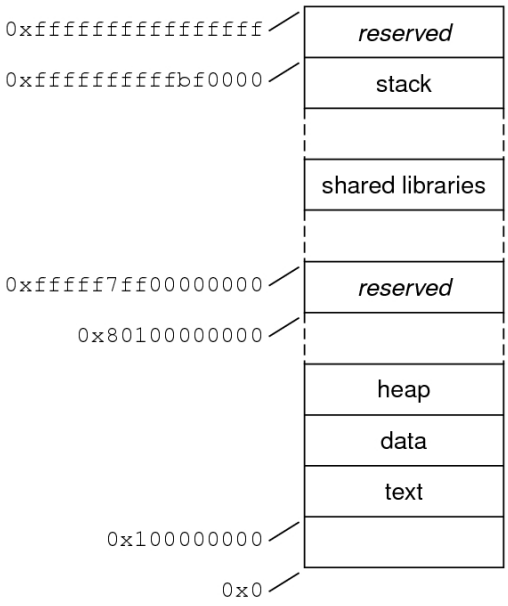


For more information about stack bias, please see the SPARC V9 ABI.

6.1.2 Address Space Layout of the SPARC V9 ABI

For 64-bit applications, the layout of the address space is closely related to that of 32-bit applications, though the starting address and addressing limits are radically different. Like SPARC V8, the SPARC V9 stack grows down from the top of the address space, while the heap extends the data segment from the bottom.

The diagram below shows the default address space provided to a 64-bit application. The regions of the address space marked as reserved might not be mapped by applications. These restrictions might be relaxed on future systems.



The actual addresses in the figure above describe a particular implementation on a particular system, and are given for illustrative purposes only.

6.1.3 Placement of Text and Data of the SPARC V9 ABI

By default, 64-bit programs are linked with a starting address of 0x100000000. The whole program is above 4 gigabytes, including its text, data, heap, stack, and shared libraries. This

helps ensure that 64-bit programs are correct by making it so the program will fault in the lower 4 gigabytes of its address space, if it truncates any of its pointers.

While 64-bit programs are linked above 4 gigabytes, you can still link them below 4 gigabytes by using a linker mapfile and the `-M` option to the compiler or linker. A linker mapfile for linking a 64-bit SPARC program below 4 gigabytes is provided in `/usr/lib/ld/sparcv9/map.below4G`.

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

6.1.4 Code Models of the SPARC V9 ABI

Different code models are available from the compiler for different purposes to improve performance and reduce code size in 64-bit SPARC programs. The code model is determined by the following factors:

- Positioning (absolute versus position-independent code)
- Code size (< 2 gigabytes)
- Location (low, middle, anywhere in address space)
- External object reference model (small or large)

The following table describes the different code models available for 64-bit SPARC programs.

TABLE 3 Code Model Descriptions: SPARCV9

Code Model	Positionability	Code Size	Location	External Object Reference Model
`abs32`	Absolute	< 2 gigabytes	Low (low 32 bits of address space)	None
`abs44`	Absolute	< 2 gigabytes	Middle (low 44 bits of address space)	None
`abs64`	Absolute	< 2 gigabytes	Anywhere	None
`pic13`	PIC	< 2 gigabytes	Anywhere	Small (<= 1024 external objects)
`pic32`	PIC	< 2 gigabytes	Anywhere	Large (<= 2**29 external objects)

Shorter instruction sequences can be achieved in some instances with the smaller code models. The number of instructions needed to do static data references in absolute code is the fewest for the `abs32` code model and the most for the `abs64` code model, while `abs44` is in the middle. Likewise, the `pic` code model uses fewer instructions for static data references than the `PIC` code model. Consequently, the smaller code models can reduce the code size and perhaps

improve the performance of programs that do not need the fuller functionality of the larger code models.

To specify which code model to use, the `-xcode=<model>` compiler option should be used. Currently, for 64-bit objects, the compiler uses the `abs44` model by default. You can optimize your code by using the `abs44` code model; you will use fewer instructions and still cover the 44-bit address space that the current UltraSPARC platforms support.

For more information about code models, see the SPARC V9 ABI and compiler documentation.

6.2 AMD64 ABI Features

64-bit applications are described by using Executable and Linking Format (ELF64), which allows large applications and large address spaces to be described completely.

Following is a list of the AMD ABI features.

- The AMD ABI allows all 64-bit instructions and 64-bit registers to be used to their full effect. Many of the new instructions are straightforward extensions of the existing i386 instruction set. There are now sixteen general purpose registers.

  Seven general purpose registers (`%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`, and `%rax`) have a well-defined role in the function call sequence which now passes arguments in registers.
  Two registers are used for stack management (`%rsp` and `%rbp`). The `%rbp` register can also be used as a general purpose register.
  Two registers are temporaries (`%r10` and `%r11`).
  Seven registers are callee-saved (`%r12`, `%r13`, `%r14`, `%r15`, `%rbx`, `%rsp`, `%rbp`).
- The basic function calling convention is different for the AMD ABI. Arguments are placed in registers. For simple integer arguments, the first arguments are placed in the `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9` registers in order. Floating-point arguments are placed from `%xmm0` to `%xmm7` registers.
- The layout of the stack is slightly different for AMD. In particular, the stack is always aligned on a 16-byte boundary immediately preceding the call instruction. AMD64 has special area on stack called red zone. This is 128-byte area beyond the location pointed to by `%rsp` is considered to be reserved and shall not be modified by signal or interrupt handlers.
- Instruction sizes are still 32-bit. Address constant generation therefore takes more instructions. The call instruction can no longer be used to branch anywhere in the address space, since it can only reach within plus or minus 2 gigabytes of `%rip`.
- Integer multiply and divide functions are now implemented completely in hardware.

- Structure passing and return are accomplished differently. Small data structures and some floating point arguments are now passed directly in registers.
- There are new PC-relative addressing modes that enable more efficient position-independent code to be generated.
- All data types are now aligned to their size.
- Many basic derived types are larger. Thus many system call interface data structures are now of different sizes.
- Two different sets of libraries exist on the system: those for 32-bit i386 applications and those for 64-bit amd64 applications.
- The AMD ABI substantially enhances floating point capabilities.

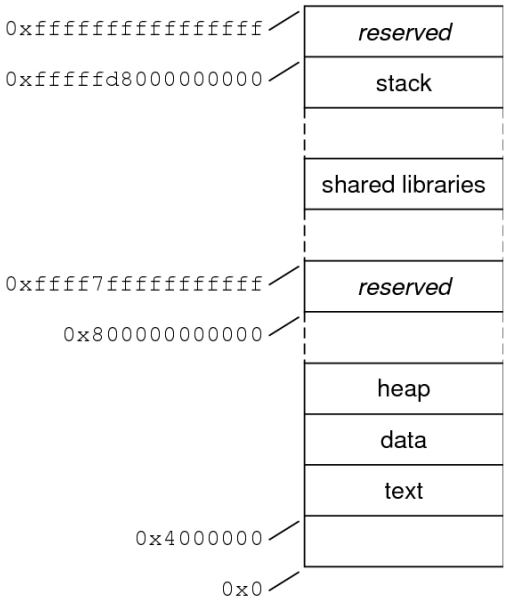
  The 64-bit ABI allows all the x87 and MMX instructions that operate on the x87 floating point registers (`%fpr0` through `%fpr7` and `%mm0` through `%mm7`) to be used.
  Additionally, the full set of SSE and SSE2 instructions that operate on the 128-bit XMM registers (`%xmm0` through `%xmm15`) can be used.
  Also the full set of AVX instructions that operate on the 256-bit `YMM` registers can be used and support for AVX instructions is optional.

For more information about amd64 ABI see, *System V Application Binary Interface, AMD64 Architecture Processor Supplement*.

6.2.1 Address Space Layout for amd64 Applications

For 64-bit applications, the layout of the address space is closely related to that of 32-bit applications, though the starting address and addressing limits are radically different. Like SPARC V9, the amd64 stack grows down from the top of the address space, while the heap extends the data segment from the bottom.

The following diagram shows the default address space provided to a 64-bit application. The regions of the address space marked as reserved might not be mapped by applications. These restrictions might be relaxed on future operating systems.



The actual addresses in the figure above describe a particular implementation on a particular system, and are given for illustrative purposes only.

6.3 Alignment Issues

There is one additional issue around the alignment of 32-bit `long long` elements in data structures; i386 applications only align `long long` elements on 32-bit boundaries, while the amd64 ABI places `long long` elements on 64-bit boundaries potentially generating wider holes in the data structures. This is different to SPARC where 32-bit or 64-bit, `long long` items were aligned on 64-bit boundaries.

The following table shows the data type alignment for the designated architectures.

TABLE 4 Data Type Alignment

Architecture	long long	double	long double
i386	4	4	4
amd64	8	8	16
sparcv8	8	8	8

Architecture	long long	double	long double
sparcv9	8	8	16

Although code might already appear to be LP64 clean on SPARC systems, the alignment differences might produce problems when copying data structures between 32-bit and 64-bit programming environments. These programming environments include device driver `ioctl` routines, `doors` routines or other IPC mechanisms. Alignment problems can be avoided by careful coding of these interfaces, and by judicious use of the `#pragma pack` or `_Pack` directives.

6.4 Interprocess Communication

The following interprocess communication (IPC) primitives continue to work between 64-bit and 32-bit processes:

- The System V IPC primitives, such as `shmop`(2), `semop`(2), `msgsnd`(2)
- The call to `mmap`(2) on shared files
- The use of `pipe`(2) between processes
- The use of `door_call`(3C) between processes
- The use of `rpc`(3NSL) between processes on the same or different machines using the external data representation described in `xdr`(3NSL)

Although all these primitives allow interprocess communication between 32-bit and 64-bit processes, you might need to take explicit steps to ensure that data being exchanged between processes is correctly interpreted by all of them. For example, two processes sharing data described by a C data structure containing variables of type `long` cannot do so without understanding that a 32-bit process views this variable as a 4-byte quantity, while a 64-bit process views this variable as an 8-byte quantity.

One way to handle this difference is to ensure that the data has exactly the same size and meaning in both processes. Build the data structures by using fixed-width types, such as `int32_t` and `int64_t`. Care is still needed with alignment. Shared data structures might need to be padded out, or repacked by using compiler directives such as `#pragma pack` or `_Pack`. See “6.3 Alignment Issues” on page 57.

A family of derived types that mirrors the system derived types is available in `<sys/types32.h>`. These types possess the same sign and sizes as the fundamental types of the 32-bit system but are defined in such a way that the sizes are invariant between the ILP32 and LP64 compilation environments.

Sharing pointers between 32-bit and 64-bit processes is substantially more difficult. Obviously, pointer sizes are different, but more importantly, while there is a 64-bit integer quantity (`long long`) in existing C usage, a 64-bit pointer has no equivalent in a 32-bit environment. In order for a 64-bit process to share data with a 32-bit process, the 32-bit process can only see up to 4 gigabytes of that shared data at a time.

The XDR routine `xdr_long(3NSL)` might seem to be a problem; however, it is still handled as a 32-bit quantity over the wire to be compatible with existing protocols. If the 64-bit version of the routine is asked to encode a `long` value that does not fit into a 32-bit quantity, the encode operation fails.

6.5 ELF and System Generation Tools

64-bit binaries are stored in files in ELF64 format, which is a direct analog of the ELF32 format, except that most fields have grown to accommodate full 64-bit applications. ELF64 files can be read by using `elf(3ELF)` APIs. For example, `elf_getarhdr(3ELF)`.

Both 32-bit and 64-bit versions of the ELF library, `elf(3ELF)`, support both ELF32 and ELF64 formats and their corresponding APIs. This allows applications to build, read, or modify both file formats from either a 32-bit or a 64-bit system (though a 64-bit system is still required to execute a 64-bit program).

In addition, Oracle Solaris provides a set of GELF (Generic ELF) interfaces that allow the programmer to manipulate both formats by using a single, common API. For more information, see the `gelf(3ELF)` man page.

All of the system ELF utilities, including `ar(1)`, `nm(1)`, `ld(1)` and `dump(1)`, have been updated to accept both ELF formats.

6.6 `/proc` Interface

The `/proc` interfaces are available to both 32-bit and 64-bit applications. 32-bit applications can examine and control the state of other 32-bit applications. Thus, an existing 32-bit debugger can be used to debug a 32-bit application.

64-bit applications can examine and control 32-bit or 64-bit applications. However, 32-bit applications are unable to control 64-bit applications, because the 32-bit APIs do not allow the full state of 64-bit processes to be described. Thus, a 64-bit debugger is required to debug a 64-bit application.

6.7 Extensions to sysinfo() System Call

The sysinfo() system call subcodes in the Oracle Solaris 10 and later versions enable applications to ascertain more information about the available instruction set architectures.

```
SI_ARCHITECTURE_32
SI_ARCHITECTURE_64
SI_ARCHITECTURE_K
SI_ARCHITECTURE_NATIVE
```

For example, the name of the 64-bit ABI available on the system (if any), is available by using the SI_ARCHITECTURE_64 subcode. For more information, see the sysinfo(2) man page.

6.8 libkvm and /dev/ksyms

Oracle Solaris OS is implemented by using a 64-bit kernel and applications that examine or modify the contents of the kernel directly must be converted to 64-bit applications and linked with the 64-bit version of libraries.

Before doing this conversion and cleanup work, you should examine why the application needs to look directly at kernel data structures in the first place. It is possible that in the time since the program was first ported or created; additional interfaces have been made available on the Oracle Solaris platform, to extract the needed data with system calls. The most common APIs are sysinfo(2), kstat(3KSTAT), sysconf(3C), getloadavg(3C), and proc(4). If these interfaces can be used instead of kvm_open(3KVM), you must use them for maximum portability. As a further benefit, most of these APIs are probably faster and might not require the same security privileges needed to access kernel memory.

The 32-bit version of libkvm returns a failure from any attempt to use kvm_open(3KVM) on a 64-bit kernel or crash dump. Similarly, the 64-bit version of libkvm returns failure from any attempt to use kvm_open(3KVM) on a 32-bit kernel crash dump.

Because the kernel is a 64-bit program, applications that open /dev/ksyms to examine the kernel symbol table directly need to be enhanced to understand ELF64 format.

The ambiguity over whether the address argument to kvm_read() or kvm_write() is supposed to be a kernel address or a user address is even worse for 64-bit applications and kernel. All applications using libkvm that are still using kvm_read() and kvm_write() should transition

to use the appropriate kvm_read(), kvm_write(), kvm_uread(), and kvm_uwrite() routines. (These routines were first made available in Solaris 2.5.)

Applications that read /dev/kmem or /dev/mem directly can still run, though any attempt they make to interpret data they read from those devices might be wrong; data structure offsets and sizes are almost certainly different between 32-bit and 64-bit kernels.

6.9 libkstat Kernel Statistics

The sizes of many kernel statistics are completely independent of whether the kernel is a 64-bit or 32-bit program. The data types exported by named kstats are self-describing, and export signed or unsigned, 32-bit or 64-bit counter data, appropriately tagged. Thus, applications using libkstat need *not* be made into 64-bit applications to work successfully with the 64-bit kernel. For more information, see the kstat(3KSTAT) man page.

Note - If you are modifying a device driver that creates and maintains named kstats, you should try to keep the size of the statistics you export invariant between 32-bit and 64-bit kernels by using the fixed-width statistic types.

6.10 Changes to stdio

In the 64-bit environment, the stdio facility allows more than 256 streams to be open simultaneously. Given the large number of open streams, iterating over every file descriptor is inefficient. To improve efficiency, you can use the closefrom() and fdwalk() API's to iterate over the open file descriptors. For more information, see closefrom(3C) and fdwalk(3C) man pages.

Note - The 32-bit stdio facility continues to have the 256 streams limit.

64-bit applications should not rely on having access to the members of the FILE data structure. Attempts to access private implementation-specific structure members directly can result in compilation errors. Existing 32-bit applications are unaffected by this change, but any direct usage of these structure members should be removed from all code.

The FILE structure has a long history, and a few applications have looked inside the structure to glean additional information about the state of the stream. Because the 64-bit version of the structure is now opaque, a new family of routines has been added to both 32-bit libc and 64-

bit `libc` to allow the same state to be examined without depending on implementation internals. For example, see the `__fbufsize`(3C) man page.

6.11 Building FOSS on Oracle Solaris Systems

To build Free and Open Source Software (FOSS) components on Oracle Solaris, you must consider the following.

- Using `pkg-config` – You can use the `pkg-config` utility to find build configuration flags. To get the appropriate flags for a 64-bit build, you must set the following environment variable before you run the `pkg-config` utility.

  ```
  PKG_CONFIG_PATH=/usr/lib/64/pkgconfig
  ```

  If you have additional third-party libraries installed in other paths, you must include the 64-bit variants of those paths in the `PKG_CONFIG_PATH` setting. For example:

  ```
  PKG_CONFIG_PATH=/usr/lib/64/pkgconfig:/opt/local/lib/64/pkgconfig
  ```

  The `pkg-config` utility will search the default path after searching the directories mentioned in the `PKG_CONFIG_PATH` setting. Therefore, the `pkg-config` utility finds the configuration files that are independent of bit size in the `/usr/share/pkgconfig` directory and the 32-bit configuration files in the `/usr/lib/pkgconfig` directory.
- Using GNU `autoconf` – You can use the GNU `autoconf` utility to find the build flags for C or C++ code. To get the appropriate flags for a 64-bit build, you must set the following in your environment variables before running the configure script.

  ```
  CFLAGS="-m64" CXXFLAGS="-m64"
  ```

  If the configure script calls `pkg-config` for library paths, you must need the `PKG_CONFIG_PATH` variable listed above. If it finds any libraries via `LDFLAGS`, adding any flags for libraries in the default paths of `/lib/64` or `/usr/lib/64`, files is not required. For other libraries in other paths you must set following flag.

  ```
  LDFLAGS="-L/opt/local/lib/64 -R/opt/local/lib/64"
  ```

6.12 Performance Issues

The following sections discuss advantages and disadvantages of 64-bit to application performance.

6.12.1 64-Bit Application Advantages

- Arithmetic and logical operations on 64-bit quantities are more efficient.
- Operations use full-register widths, the full-register set, and new instructions.
- Parameter passing of 64-bit quantities is more efficient.
- Parameter passing of small data structures and floating point quantities is more efficient.
- Additional integer and floating point registers.
- For amd64, PC-relative addressing modes for more efficient position-independent code.

6.12.2 64-Bit Application Disadvantages

- 64-bit applications require more stack space to hold the larger registers.
- Applications have a bigger cache footprint from larger pointers.
- Address formation may take additional or larger instructions.

6.13 System Call Issues

The following sections discuss system call issues.

6.13.1 `EOVERFLOW` Indicates System Call Issue

The `EOVERFLOW` return value is returned from a system call whenever one or more fields of the data structure used to pass information out of the kernel is too small to hold the value.

A number of 32-bit system calls now return `EOVERFLOW` when faced with large objects on the 64-bit kernel. While this was already true when dealing with large files, the fact that `daddr_t`, `dev_t`, `time_t`, and its derivative types `struct timeval` and `timespec_t` now contain 64-bit quantities might mean more `EOVERFLOW` return values are observed by 32-bit applications.

6.13.2 `ioctl()` Does Not Type Check at Compile Time

Some ioctl(2) calls have been rather poorly specified in the past. Unfortunately, `ioctl()` is completely devoid of compile-time type checking; therefore, it can be a source of bugs that are difficult to track down.

Consider two `ioctl()` calls – one that manipulates a pointer to a 32-bit quantity `(IOP32)`, the other that manipulates a pointer to a long quantity `(IOPLONG)`.

The following code sample works as part of a 32-bit application when it is compiled:

```
int a, d;
long b;
...
if (ioctl(d, IOP32, &b) == -1)
  return (errno);
if (ioctl(d, IOPLONG, &a) == -1)
  return (errno);
```

In a 64-bit application, the preceding `ioctl()` calls succeed but do not work correctly:

- The first `ioctl()` call passes a container that is too big, and on a big-endian implementation, the kernel will copy in or copy out from the wrong part of the 64-bit word. Even on a little-endian implementation, the container probably contains stack garbage in the upper 32 bits.
- The second `ioctl()` call will copy in or copy out too much, either reading an incorrect value, or corrupting adjacent variables on the user stack.

♦ ♦ ♦ APPENDIX A

Changes in Derived Types

The current 32-bit compilation environment is identical to historical Oracle Solaris OS releases with respect to derived types and their sizes. In the 64-bit compilation environment, some changes in derived types are necessary. These changed derived types are highlighted in the tables that follow.

Note that although the 32-bit and 64-bit compilation environments differ, the same set of headers is used for both, with the appropriate definitions determined by the compilation options. To understand the options available to an application developer, consider the `_ILP32` and `_LP64` feature test macros:

`_ILP32` feature test macro

Specifies the ILP32 data model where `int`, `long` and pointers are 32-bit values. By itself, the use of this macro makes visible those derived types and sizes identical to historical Oracle Solaris implementations. This is the default compilation environment when building 32-bit applications. It ensures complete binary and source compatibility for both C and C++ applications.

`_LP64` feature test macro

Specifies the `_LP64` data model where `int` are 32-bit values and `long` and pointers are 64-bit values. `_LP64` is defined by default when compiling in 64-bit mode. Other than making sure that either `sys/types.h` or `sys/feature_tests.h` is included in source in order to make visible the `_LP64` definition, the developer needs to do nothing else.

Note - The default compilation mode depends on the compiler being used. To determine whether your compiler produces 64-bit or 32-bit code by default, refer to the compiler documentation.

The following examples illustrate the use of feature test macros so that the correct definitions are visible, depending on the compilation environment.

EXAMPLE 2 `size_t` Definition in `_LP64`

```
#if defined(_LP64)
typedef ulong_t size_t;   /* size of something in bytes */
#else
typedef uint_t size_t;    /* (historical version) */
#endif
```

When building a 64-bit application with the definition in this example, `size_t` is a `ulong_t`, or `unsigned long`, which is a 64-bit value in the LP64 model. In contrast, when building an 32-bit application, `size_t` is defined as a `uint_t`, or `unsigned int`, a 32-bit value in either in the ILP32 or the LP64 models.

EXAMPLE 3 `uid_t` Definition in `_LP64`

```
#if defined(_LP64)
typedef int     uid_t;          /* UID type             */
#else
typedef long    uid_t;          /* (historical version) */
#endif
```

In the preceding examples, the same result would have been obtained had the `ILP32` type representation been identical to the `LP64` type representation. For example, if in the 32-bit application environment, `size_t` was changed to an `ulong_t`, or `uid_t` was changed to an `int`, these would still represent 32-bit quantities. However, retaining the historical type representation ensures consistency within 32-bit C and C++ applications, as well as complete binary and source compatibility with prior releases of the Oracle Solaris operating environment.

Table 5, “Differences Between Derived Types – General,” on page 66 lists the derived types that have changed. When building a 32-bit application, the derived types available to the developer match those in the `_ILP32` column. When building a 64-bit application, the derived types match those listed in the `_LP64` column. All of these types are defined in `<sys/types.h>`, with the exception of the `wchar_t` and `wint_t` types, which are defined in `<wchar.h>`.

When reviewing these tables, remember that in the 32-bit environment, `int`, `long`, and pointers are 32-bit quantities. In the 64-bit environment, `int` are 32-bit quantities while `long` and pointers are 64-bit quantities.

TABLE 5 Differences Between Derived Types – General

Derived Types	_ILP32	_LP64
blkcnt_t	longlong_t	long
id_t	long	int
major_t	ulong_t	uint_t

Derived Types	_ILP32	_LP64
minor_t	ulong_t	uint_t
mode_t	ulong_t	uint_t
nlink_t	ulong_t	uint_t
paddr_t	ulong_t	*not defined*
pid_t	long	int
ptrdiff_t	int	long
size_t	uint_t	ulong_t
ssize_t	int	long
uid_t	long	int
wchar_t	long	int
wint_t	long	int

Table 6, “Differences Between Derived Types Specific to Large Files,” on page 67 lists the derived types specific to the Large Files compilation environment. These types are only defined if the feature test macro `_LARGEFILE64_SOURCE` is defined.

TABLE 6 Differences Between Derived Types Specific to Large Files

Derived Types	_ILP32	_LP64
blkcnt64_t	longlong_t	blkcnt_t
fsblkcnt64_t	u_longlong_t	blkcnt_t
fsfilcnt64_t	u_longlong_t	fsfilcnt_t
ino64_t	u_longlong_t	ino_t
off64_t	longlong_t	off_t

Table 7, “Changed Derived Types – `FILE_OFFSET_BITS` Value,” on page 67 lists the changed derived types with respect to the value of `_FILE_OFFSET_BITS`. You cannot compile an application with both `_LP64` defined and `_FILE_OFFSET_BITS==32`. By default, if `_LP64` is defined, `_FILE_OFFSET_BITS==64`. If `_ILP32` is defined, and `_FILE_OFFSET_BITS` is not defined, then by default, `_FILE_OFFSET_BITS==32`. These rules are defined in the `sys/feature_tests.h` header file.

TABLE 7 Changed Derived Types – `FILE_OFFSET_BITS` Value

Derived Types	_ILP32 _FILE_ OFFSET_BITS ==32	_ILP32 _FILE_ OFFSET_BITS ==64	_LP64 _FILE_ OFFSET_BITS==64
ino_t	ulong_t	u_longlong_t	ulong_t

Derived Types	_ILP32 _FILE_ OFFSET_BITS ==32	_ILP32 _FILE_ OFFSET_BITS ==64	_LP64 _FILE_ OFFSET_BITS==64
blkcnt_t	long	longlong_t	long
fsblkcnt_t	ulong_t	u_longlong_t	ulong_t
fsfilcnt_t	ulong_t	u_longlong_t	ulong_t
off_t	long	longlong_t	long

♦ ♦ ♦ APPENDIX B

Frequently Asked Questions (FAQ)

Is Oracle Solaris a 64-bit operating system?

Yes. Oracle Solaris 11 and any later version is a 64-bit only operating system. However, you can run your 32-bit applications on the 64-bit Oracle Solaris 11 OS.

Do all my applications need to be 64-bit?

Oracle Solaris 11 and later versions is a 64-bit only operating system. Therefore, any new application that you develop should be a 64-bit application.

Can I run the 64-bit version of the operating system on 32-bit hardware?

No. It is not possible to run the 64-bit operating system on 32-bit hardware. The 64-bit operating system requires 64-bit MMU and CPU hardware.

Do I need to change my 32-bit application if I plan to run that application on a system with the 64-bit operating system?

Most applications can remain 32-bit and still execute on 64-bit operating system without requiring code changes or recompilation. However, if you want 32-bit applications to run after January 2038 and get all the benefits of a 64-bit OS, you must convert your 32-bit applications to 64-bit applications.

If your application uses `libkvm`(3LIB), it must be recompiled as 64-bit, to execute on 64-bit operating system. If your application uses `/proc`, it might need to be recompiled as 64-bit; otherwise it cannot understand a 64-bit process. This is because the existing interfaces and data structures that describe the process are not large enough to contain the 64-bit values involved.

Can I build a 32-bit application on a system running the 64-bit operating system?

Yes. The current default compilation mode is 32-bit. However, the future releases of Oracle Solaris will have 64-bit as the default compilation mode.

Can I combine 32-bit libraries and 64-bit libraries when building and linking applications?

No. 32-bit applications must link with 32-bit libraries and 64-bit applications with 64-bit libraries. Attempts to build or link with the wrong version of a library will result in an error.

What are the sizes of floating point data types in the 64-bit implementation?

The *only* types that have changed are `long` and `pointer`. See Table 1, “Data Type Sizes in Bits,” on page 24.

What about `time_t`?

The `time_t` type remains a `long` quantity. In the 64-bit environment, `time_t` is a 64-bit quantity. Thus, 64-bit applications will be year 2038 safe.

What is the value of `uname(1)` on a system running the 64-bit Solaris operating environment?

The output of the `uname -p` command is unchanged.

Index

C

D

E

F

G

H

I

K

L

M

O

P

S

T

U

V