Solaris 64-bit Developer's Guide

Part No: 816–5138–10
January 2005

ORACLE

110417@25097

Contents

Preface

The capabilities of the Solaris operating environment continue to expand to meet customer needs. The Solaris operating system was designed to fully support both the 32-bit and 64-bit architectures. The Solaris operating environment provides an environment for building and running 64-bit applications that can use large files and large virtual address spaces. At the same time, the Solaris operating environment continues to provide maximum source compatibility, maximum binary compatibility, and interoperability for 32-bit applications. In fact, most of the system commands that run and have been built on the Solaris 64-bit implementation are 32-bit programs.

Note – This Solaris release supports systems that use the SPARC and x86 families of processor architectures: UltraSPARC, SPARC64, AMD64, Pentium, and Xeon EM64T. The supported systems appear in the *Solaris 10 Hardware Compatibility List* at `http://www.sun.com/bigadmin/hcl`. This document cites any implementation differences between the platform types.

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. For supported systems, see the *Solaris 10 Hardware Compatibility List*.

The major differences between the 32-bit and the 64-bit application development environments are that 32-bit applications are based on the ILP32 data model, where `int`s, `long`s, and pointers are 32 bits, while 64-bit applications are based on the LP64 model, where `long`s and pointers are 64 bits and the other fundamental types are the same as in ILP32.

Most applications can remain as 32-bit programs with no changes required. Conversion is necessary only if the 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 `/proc` 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

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.

Who Should Use This Book

This document is written for C and C++ developers and provides guidance on how to determine whether an application is 32-bit or 64-bit. This document provides

- A list of the similarities and differences between the 32-bit and 64-bit application environments
- An explanation of how to write code that is portable between the two environments
- A description of the tools provided by the operating system for developing 64-bit applications

How This Book Is Organized

This book is organized into the following chapters.

- Chapter 1, "64-bit Computing," describes the motivation behind 64–bit computing and gives an overview of the benefits of 64–bit applications.
- Chapter 2, "When to Use 64-bit," explains the differences between the Solaris 32-bit and 64-bit build and runtime environments. The information is written to help the application developer determine if and when converting code to be 64-bit safe is appropriate.
- Chapter 3, "Comparing 32-bit Interfaces and 64–bit Interfaces," focuses on the similarities between 32-bit applications and 64-bit applications as well as the 64-bit interfaces.
- Chapter 4, "Converting Applications," describes how to convert current 32–bit code to 64-bit safe code and the tools available for making this process easier. The focus of this chapter is on writing portable code. The information applies to converting existing applications or writing new applications that are capable of running in both 32-bit and 64-bit environments.
- Chapter 5, "The Development Environment," focuses on the build environment, including headers, compilers, and libraries, as well as packaging guidelines and debugging tools.
- Chapter 6, "Advanced Topics," is an overview of 64–bit systems programming, the ABI, and some performance issues.
- Appendix A, "Changes in Derived Types," highlights many of the derived types that have changed in the 64-bit application development environment.

- Appendix B, "Frequently Asked Questions (FAQs)," provides answers to the most commonly asked questions about the 64-bit implementation and application development environment.

Related Books

For further reading, the following texts are recommended:

- American National Standard for Information Systems Programming Language - C, ANSI X3.159-1989
- SPARC Architecture Manual, Version 9, SPARC International
- SPARC Compliance Definition, Version 2.4, SPARC International
- *Large Files in Solaris: A White Paper*, Part No: 96115-001
- *Solaris 10 Reference Manual*
- *Writing Device Drivers*, Part No: 816–4854
- *Sun Studio 10: C User's Guide*, Part No: 819-0494-10

Accessing Sun Documentation Online

The docs.sun.com Web site enables you to access Sun technical documentation online. You can browse the docs.sun.com archive or search for a specific book title or subject. The URL is http://docs.sun.com.

What Typographic Conventions Mean

The following table describes the typographic changes that are used in this book.

TABLE P–1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`** `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type **`rm`** *filename*.

TABLE P–1 Typographic Conventions *(Continued)*

Typeface or Symbol	Meaning	Example
AaBbCc123	Book titles, new words or new terms, or words to be emphasized.	Read Chapter 6 in *User's Guide*. These are called *class* options. Do *not* save changes yet.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

◆ ◆ ◆ 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 of existing applications continue to grow. Today, certain classes of applications need to exceed the 4 Gigabyte address space limitations of 32-bit systems. Examples of applications that exceed the 4 Gigabyte address space include

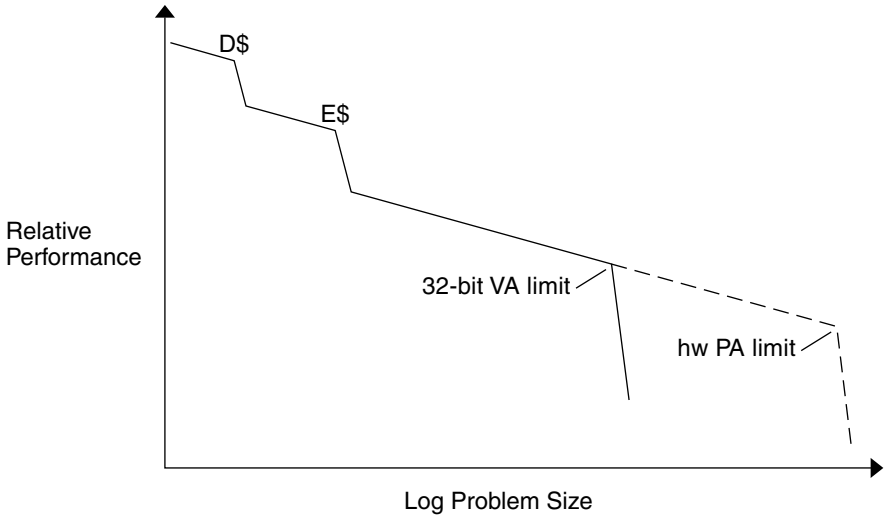
- Various database applications, particularly those applications that perform data mining
- Web caches and Web search engines
- Components of CAD and CAE simulation and modeling tools
- Scientific computing

The desire to make these applications and other large applications run efficiently has been the primary impetus for the development of 64-bit computing.

Getting Past the 4 Gigabyte Barrier

The diagram in Figure 1–1 plots typical performance against problem size for an application running on a machine with a large amount of physical memory installed. 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 1–1 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 more than 4 Gigabytes of physical memory. High-end desktop machines are following the same trend, but no single 32-bit program can 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, then 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 machine.

It might seem obvious, but for a broad class of applications, the ability to handle larger problems directly in primary memory *is* the major performance benefit of 64-bit machines.

- 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.

Beyond Large Address Spaces

Other compelling reasons why you might want to create 64-bit applications include:

- You need to perform a lot of computation on 64-bit integer quantities that use the wider 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.
- You want to obtain the performance benefits of the 64-bit instruction set, such as improved calling conventions and full use of the register set.

◆ ◆ ◆ CHAPTER 2

When to Use 64-bit

For application developers, the major difference between the Solaris 64-bit and 32-bit operating environments is the C data type model used. The 64–bit version uses the LP64 model where `longs` and pointers are 64–bits. All other fundamental data types remain the same as in the 32-bit implementation, which is based on the ILP32 model. In the ILP32 model, `ints`, `longs`, and pointers are 32-bit quantities. These models are explained in greater detail in Chapter 3, "Comparing 32-bit Interfaces and 64–bit Interfaces."

Few applications really *require* conversion. Most applications can remain as 32-bit applications and still run on the 64-bit operating system without requiring any code changes or recompilation. In fact, 32-bit applications that do not require 64-bit capabilities should probably remain 32-bit to maximize portability.

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

You might need to convert applications with these characteristics:

- 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

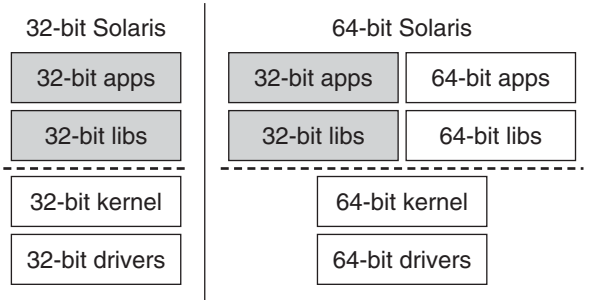
Some specific interoperability issues require code changes. Similarly, if your application uses files that are larger than 2 gigabytes, consider conversion to a 64-bit application instead of using the large file APIs directly.

These items are explained further in the sections that follow.

Major Features

To explain the dual 32-bit and 64-bit support in the Solaris operating environment, the following figure shows the stacks side-by-side. The system on the left supports only 32–bit libraries and applications on top of a 32-bit kernel that uses 32-bit device drivers. The system on the right supports the same 32–bit applications and libraries as on the left. This system also supports 64-bit libraries and applications *simultaneously* on top of a 64-bit kernel that uses 64-bit device drivers.

FIGURE 2–1 The Solaris 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

Large Virtual Address Space

In the 64-bit environment, a process can have up to 64–bits 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 – Because of hardware restrictions, some platforms might not support the full 64–bits of address space.

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 a 64-bit program. A 64-bit program takes full advantage of 64-bit capabilities with a coherent set of interfaces.

64–bit Arithmetic

64-bit arithmetic has long been available in previous 32-bit Solaris releases. However, the 64-bit implementation now uses the full 64-bit machine 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.

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 that concern future events, such as mortgages, might require the expanded time capability.

Interoperability Issues

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

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

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 programs.

`/proc` Restrictions

A 32-bit program that uses `/proc` is able to look at 32-bit processes, but is not able to 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 quantities that are involved. Such programs need to 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.

64–bit Libraries

32-bit applications are required to link with 32-bit libraries, and 64-bit applications are required to link with 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.

Estimating the Effort of Conversion

After you've 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 "Getting Past the 4 Gigabyte Barrier" on page 11, most 32-bit applications run unchanged in the Solaris 64-bit operating environment. Some applications might only need to be recompiled as 64-bit applications, others need to be converted. This chapter is directed at developers who have determined that their application needs to be recompiled or converted to 64-bit, based on the items discussed in "Getting Past the 4 Gigabyte Barrier" on page 11.

Application Programming Interfaces

The 32-bit application programming interfaces (APIs) supported in the 64-bit operating environment are the 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.

The default 64-bit APIs are basically the UNIX 98 family of APIs. Their specification is written in terms of derived types. The 64-bit versions are obtained by expanding some of the derived types to 64-bit quantities. Correctly written applications that use these APIs are portable in source form between the 32-bit environment and 64-bit environment. The UNIX 2001 API family is also available in Solaris 10, see `standards`(5).

Application Binary Interfaces

The SPARC V8 ABI is the existing processor-specific Application Binary Interface (ABI) on which the 32-bit SPARC version of the Solaris implementation is based. The SPARC V9 ABI extends the SPARC V8 ABI to support 64-bit operations and defines new capabilities for this extended architecture. See "SPARC V9 ABI Features" on page 49 for additional information.

The i386 ABI is the processor-specific ABI on which the 32-bit version of Solaris (Intel Platform Edition) is based.

For the Solaris 10 release, the amd64 ABI is the processor-specific ABI on which the 64-bit version of Solaris on x86 systems is based. The amd64 ABI supports 64-bit operations and defines new capabilities for the new architecture. Programs using the 64-bit ABI might perform better than their 32-bit counterparts. Processors that support the amd64 ABI also support the i386 ABI. See "AMD64 ABI Features" on page 53 for additional information.

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.

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`. See "Getting Past the 4 Gigabyte Barrier" on page 11 for more information.

Application Source Code

Source level compatibility has been maintained for 32-bit applications. For 64-bit applications, the principal changes that have been made are with respect to the derived types used in the application programming interface. Applications that use the derived types and interfaces correctly are source compatible for 32-bit, and make the transition to 64-bit more easily.

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 need to 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, the fundamental driver model and the interfaces supported by the DDI do not change. The principal work is to clean up the code to be correct in an LP64 environment. See the *Writing Device Drivers* manual for more information.

Which Solaris Operating Environment Are You Running?

The Solaris operating environment supports two first-class ABIs simultaneously. In other words, two separate, fully functional system call paths connect into the 64–bit kernel. Two sets of libraries support applications.

The 64-bit operating system can run on only 64-bit CPU hardware, while the 32-bit version can run on either 32-bit CPU hardware or 64-bit CPU hardware. Because the Solaris 32-bit and 64-bit operating environments look very similar, it might not be immediately apparent which version is running on a particular hardware platform.

The easiest way to determine which version is running on your system is to use the `isainfo` command. This new command prints information about the application environments supported on the system.

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

```
% isainfo -v
64-bit sparcv9 applications
32-bit sparc applications
```

When the same command is run on an x86 system runnig the 32–bit Solaris operating system:

```
% isainfo -v
32-bit i386 applications
```

When the same command is run on an x86 system running the 64–bit Solaris operating system:

```
% isainfo -v
64-bit amd64 applications
32-bit i386 applications
```

Note – Not all x86 systems are capable of running the 64-bit kernel. In this case, if the system is running the in Solaris operating environment, the kernel is running in 32-bit mode

---

One useful option of the `isainfo(1)` 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 Solaris"
Welcome to 64-bit Solaris
```

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

A related command, `isalist(1)`, is more suited for use in shell scripts. `isalist` can be used to print the complete list of supported instruction sets on the platform. However, as the number of instruction set extensions increases, the limitations of a list of all subsets has become apparent. Users are advised to not rely upon this interface in the future.

Users who create libraries that depend upon 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.

```
% 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

Two basic issues that regard conversion arise for applications developers:

- Data type consistency and the different data models
- Interoperation between applications that use different data models

Trying to maintain a single source with as few `#ifdefs` as possible is usually better than trying to maintain multiple source trees. This chapter provides guidelines for writing code that works correctly in both 32-bit environments and 64-bit environments. At best, the conversion of current code might require only a recompilation and relinking with the 64-bit libraries. However, for those cases where code changes are required, this chapter discusses the tools that help make conversion easier.

Data Model

As stated previously, the biggest difference between the 32-bit environment and 64-bit environment is the change in two fundamental data types.

The C data-type model used for 32-bit applications is the ILP32 model, so named because `int`s, `long`s, and pointers are 32-bit. The LP64 data model is the C data-type model for 64-bit applications. This model was agreed upon by a consortium of companies across the industry. LP64 is so named because longs and pointers grow to 64-bit quantities. The remaining C types `int`, `short`, and `char` are the same as in the ILP32 model.

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 -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 -xarch=generic64 -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 environment is designed to maximize portability, that is, to create 32–bit applications.

The standard relationship between C integral types still holds true.

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

Table 4–1 lists the basic C types, and their corresponding sizes in bits in the data type models for both LP32 and LP64.

TABLE 4–1 Data Type Sizes in Bits

C data type	ILP32	LP64
`char`	8	unchanged
`short`	16	unchanged
`int`	32	unchanged
`long`	32	**64**
`long long`	64	unchanged

TABLE 4–1 Data Type Sizes in Bits *(Continued)*

C data type	ILP32	LP64
`pointer`	32	**64**
`enum`	32	unchanged
`float`	32	unchanged
`double`	64	unchanged
`long double`	128	unchanged

Some older 32-bit applications use `int`, `long`, and pointer types interchangeably. The size of `long`s and pointers grow in the LP64 data model. You need to be aware that this change alone can cause many 32-bit to 64-bit conversion problems.

In addition, declarations and casts become very important in showing what is intended. 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 is intended, you might need to declare the types of constants. Casts might also be needed in expressions to make certain that the expression is evaluated the way that you intended. Correct evaluation of expressions is particularly crucial in the case of sign extension, where explicit casting might be essential to achieve the intended effect.

Other problems arise with built-in C operators, format strings, assembly language, and compatibility and interoperability.

The rest of this chapter advises you how to overcome these problems by:

- Explaining the problems outlined above in more detail
- Describing some of the derived types and include files that are useful to make code safe for both 32-bit and 64-bit
- Describing the tools available for helping to make code 64-bit safe
- Providing general rules for making code portable between the 32-bit and 64-bit environments

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, as well as in Appendix A, “Changes in Derived Types.”

Feature Test Macros

An application source file that includes <sys/types.h> makes the definitions of the programming model symbols, _LP64 and _ILP32, available through inclusion of <sys/isa_defs.h>.

For information about preprocessor symbols (_LP64 and _ILP32) and macros (_LITTLE_ENDIAN and _BIG_ENDIAN6), see types(3HEAD).

Derived Types

Using the system derived types helps make code 32-bit and 64-bit safe, since the derived types themselves are safe for both the ILP32 and LP64 data models. In general, using derived types to allow for change is good programming practice. Should the data model change in the future, or when porting to a different platform, only the system derived types need to change rather than the application.

<sys/types.h> File

The <sys/types.h> header contains a number of basic derived types that should be used whenever appropriate. In particular, the following are of special interest:

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

The use of some of these types is explained in more detail later in this chapter under "Guidelines for Converting to LP64" on page 31.

<inttypes.h> File

The include file <inttypes.h> 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 is part of an ANSI C proposal and tracks the ISO/JTC1/SC22/WG14 C committee's working draft for the revision of the current ISO C standard, ISO/IEC 9899:1990 Programming Language – C.

The basic features provided by <inttypes.h> are:

- A set of fixed-width integer types
- uintptr_t and other helpful types
- Constant macros
- Limits
- Format string macros

These are discussed in more detail in the sections that follow.

Fixed-Width Integer Types

The fixed-width integer types provided by <inttypes.h> include both signed and unsigned integer types, such as int8_t, int16_t, int32_t, int64_t, 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.

These fixed-width types should *not* be used indiscriminately. For example, int can continue to be used for such things as loop counters and file descriptors, and long can be used for array indices. On the other hand, you should use fixed-width types for explicit binary representations of:

- On-disk data
- Over-the-wire data
- Hardware registers
- Binary interface specifications (that have explicitly sized objects or involve sharing or communication between 32–bit and 64–bit programs)
- Binary data structures (that are used by 32–bit and 64–bit programs through shared memory, files, and so on)

`uintptr_t` and Other Helpful Types

Other useful types provided by `<inttypes.h>` include signed and unsigned integer types large enough to hold a pointer. These are given as `intptr_t` and `uintptr_t`. In addition, `intmax_t` and `uintmax_t` are defined to be the longest (in bits) signed and unsigned integer types available.

Using the `uintptr_t` type as the integral type for pointers is a better option than using 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, the use of the `uintptr_t` requires only the definition of `uintptr_t` to change when a different data model is used. This makes it portable to many other systems. It 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 do address arithmetic. They should be used 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.

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(1)` appends `ll` to the constant `1` 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 "Guidelines for Converting to LP64" on page 31.

Limits Defined by `<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 are defined. These include `INTMAX_MIN` and `INTMAX_MAX` and their corresponding unsigned versions.

Format String Macros

Macros for specifying the `printf` and `scanf` format specifiers are also provided in `<inttypes.h>`. Essentially, these macros prepend the format specifier with an `l` or `ll` to specify the argument as a `long` or `long long`, given the number of bits in the argument, which is built into the name of the macro.

Macros for `printf(3C)` format specifiers exist for printing 8-bit, 16-bit, 32-bit, and 64-bit integers, the smallest integer types, and the biggest integer types, in decimal, octal, unsigned, and hexadecimal. For example, printing a 64–bit integer in hexadecimal notation:

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

Similarly, there are macros for `scanf(3C)` format specifiers for reading 8-bit, 16-bit, 32-bit, and 64-bit integers and the biggest integer type in decimal, octal, unsigned, and hexadecimal. For example, reading an unsigned 64–bit decimal integer:

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

Do not use these macros indiscriminately. They are best used in conjunction with the fixed-width types. Refer to the section "Fixed-Width Integer Types" on page 27 for more details.

Tools Support

The `lint` program, available with the Sun Studio 10 compiler, can detect potential 64-bit problems and is useful in making code 64-bit safe. In addition, the `-v` option to the C compiler can be very helpful. It tells the compiler to perform additional and stricter semantic checks. It also enables certain `lint`-like checks on the named files.

For more information about the capabilities of the C compilers and `lint`, see the *Sun Studio 10: C User's Guide*.

`lint` for 32–bit and 64–bit Environments

`lint` can be used on both 32-bit and 64-bit code. Use the `-errchk=longptr64` option for code that is intended to be run in both 32–bit and 64–bit environments. The `-errchk=longptr64` option checks portability to an environment in which the size of long integers and pointers is 64 bits and the size of plain integers is 32 bits.

The `-Xarch=v9` option should be used to `lint` code intended to be run in the 64–bit SPARC environment. Use the `-errchk=longptr64` option together with the `-Xarch=v9` option to generate warnings about potential 64–bit problems for code to be run on 64–bit SPARC.

Starting with the Solaris 10 release, the `-Xarch=amd64` option should be used to `lint` code intended to be run in the 64–bit AMD environment.

Note – The `-D__sparcv9` option to `lint` is no longer necessary and should not be used.

For a description of `lint` options, see the *Sun Studio 10: C User's Guide*.

When warnings are generated, `lint(1)` prints the line number of the offending code, a warning message that describes the problem, and notes whether a pointer was involved. It can also indicate the sizes of types involved. The fact that a pointer is involved and the size of the types can be useful in finding specific 64-bit problems and avoiding the pre-existing problems between 32-bit and smaller types.

Note – Though `lint` gives warnings about potential 64-bit problems, it cannot detect all problems. You must remember that not all warnings generated by `lint` are true 64-bit problems. In many cases, code that generates a warning can be intentional and correct for the application.

The sample program and `lint(1)` output below 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 only if the constant expression will 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. An example might be in the case of casts and assignments. Exercise extreme care when using the `/*LINTED*/` comment because it can mask real problems. Refer to the *Sun Studio 10: C User's Guide* or the `lint(1)` man page for more information.

Guidelines for Converting to LP64

When using `lint(1)`, remember that not all problems result in `lint(1)` warnings, nor do all `lint(1)` 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(1)` warnings are shown.

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

Since `int`s and pointers are the same size in the ILP32 environment, a lot of code relies on this assumption. Pointers are often cast to `int` or `unsigned int` for address arithmetic. Instead, pointers could be cast to `long` because `long` and pointers are the same size in both ILP32 and LP64 worlds. Rather than explicitly using `unsigned long`, use `uintptr_t` because it expresses the intent more closely and makes the code more portable, insulating it against future changes. For example,

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

produces the warning:

```
warning: conversion of pointer loses bits
```

Using the following code will produce the clean results:

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

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

Because `int`s and `long`s 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–bits in the ILP32 data model, in the LP64 data model, a `long` is 64–bits. For example,

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

produces the warning:

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

Sign Extension

Unintended sign extension is a common problem when converting to 64–bits. It is hard to detect before the problem actually occurs because `lint(1)` does not warn you about it. 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 ANSI C. The conversion rules that seem to cause the most sign extension problems between 32-bit and 64-bit integral values are:

1. Integral promotion

   A `char`, `short`, enumerated type, or bit-field, whether signed or unsigned, can be used in any expression that calls for an `int`. If an `int` can hold all possible values of the original type, the value is converted to an `int`. Otherwise, it is converted to an `unsigned int`.

2. Conversion between signed and unsigned integers

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

For a more detailed discussion of the conversion rules, refer to the ANSI C standard. Also included in this standard are useful rules for ordinary arithmetic conversions and integer constants.

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

EXAMPLE 4–1 `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:

1. `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.
2. 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 the signed and unsigned integer promotion rule. The sign extension occurs when it is converted from an `int` to a `long`.

```
% cc -o test64 -xarch=v9 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 test32 test.c
% ./test32
addr 0x80000000
addr 0x80000000
%
```

Use Pointer Arithmetic Instead of Address Arithmetic

In general, using pointer arithmetic works better than address arithmetic because pointer arithmetic is independent of the data model, whereas address arithmetic might not be. It usually leads to simpler code as well. For example,

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

produces the warning:

```
warning: conversion of pointer loses bits
```

The following code will produce clean results:

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

Repacking a Structure

Extra padding may be added to a structure by the compiler to meet alignment requirements as `long` and pointer fields grow to 64 bits for LP64. For both the SPARCV9 ABI and the amd64 ABI, all types of structures are aligned to at least the size of the largest quantity within them. A simple rule for repacking a structure is to move the `long` and pointer fields to the beginning of the structure and rearrange the rest of the fields—usually, but not always, in descending order of size, depending on how well they can be packed. For example,

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

For better results, use:

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

Note – The alignment of fundamental types changes between the i386 and amd64 ABIs. See "Alignment Issues" on page 55.

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;
```

should be:

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

Specify Constant Types

A loss of data can occur in some constant expressions because of lack of precision. These types of problems are very hard to find. Be explicit about specifying the type(s) in your constant expressions. Add some combination of `{u,U,l,L}` to the end of each integer constant to specify its type. You might also use casts to specify the type of a constant expression. For example,

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

should be:

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

Beware of Implicit Declaration

For some compilation modes, the compiler might assume the type `int` for any function or variable that is used in a module and not defined or declared externally. Any `long`s and pointers used in this way are truncated by the compiler's implicit `int` declaration. The appropriate `extern` declaration for a function or variable should be placed in a header and not in the C

module. The header should then be included by any C module that uses the function or variable. In the case of a function or variable defined by the system headers, the proper header should still be included in the code.

For example, because `getlogin()` is not declared, the following code:

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

produces the warnings:

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

For better results, use::

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

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

`sizeof` is an `unsigned long`

In the LP64 environment, `sizeof` has the effective type of `size_t` which is implemented as an `unsigned long`. Occasionally, `sizeof` is passed to a function expecting an argument of type `int`, or is assigned or cast to an `int`. In some cases, this truncation might cause loss of data. For example,

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

produces the warnings:

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

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.

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);
```

produces the warning:

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

Use the following code to produce clean results:

```
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);
```

Other Considerations

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

Derived Types That Have Grown in Size

A number of derived types have changed so they represent 64-bit quantities in the 64-bit application environment. This change does not affect 32-bit applications; however, any 64-bit applications that consume or export data described by these types need to be reevaluated for correctness. An example of this is in applications that directly manipulate the `utmpx(4)` files. For correct operation in the 64-bit application environment, you should *not* attempt to directly access these files. Instead, you should use the `getutxent(3C)` and related family of functions.

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

Check for Side Effects of Changes

One problem to be aware of is that a type change in one area might result in an unexpected 64-bit conversion in another area. For example, in the case of a function that previously returned an `int` and now returns an `ssize_t`, all the callers need to be checked.

Check Whether Literal Uses of `long` Still Make Sense

Because a `long` is 32 bits in the ILP32 model and 64 bits in the LP64 model, there might be cases where what was previously defined as a `long` is neither appropriate nor necessary. In this case, it might be possible to use a more portable derived type.

Related to this, a number of derived types might have changed under the LP64 data model for the reason stated above. 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."

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. In the headers, these would be distinguishable by the use of the `_LP64` or `_ILP32` feature test macros. Similarly, code that is to work in 32-bit and 64-bit environments might also need to utilize the appropriate `#ifdefs`, depending on the compilation mode.

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.

Checklist for Getting Started

Assuming you need to convert your code to 64-bit, the following checklist might be helpful:

- Read this entire document with an emphasis on the "Guidelines for Converting to LP64" 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(1)` using the `-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(1)` 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 version on the 32-bit operating system, and the 64-bit version on the 64-bit operating system. You could include testing the 32-bit version on the 64-bit operating system, but this is not necessary.

Sample 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 -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 -xarch=generic64 -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 environment is designed to maximize portability, that is, to create 32–bit applications.

◆ ◆ ◆ CHAPTER 5

The Development Environment

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

Before you begin, though, it is important to determine whether your installed version of the operating system is 32-bit or 64-bit. If you have come this far, the assumption is that you are running on the 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." Even if you are using the 32-bit operating environment, you can still build your 64-bit applications, provided you have the system header files and 64–bit libraries on your system.

Build Environment

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

Header Files

A single set of system headers supports both 32-bit and 64-bit compilation environments. You do not need to specify a different include path for the 64-bit compilation environment.

To better understand the changes made to the headers for support of the 64-bit environment, you should understand the various definitions in the header `<sys/isa_defs.h>`. 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 symbols `__sparcv8` and `__sparcv9` are mutually exclusive and are only relevant when the symbol `__sparc` is defined.

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

Symbol	Description
`_ILP32`	The data model where sizes of `int`, `long`, and pointer are all 32 bits.
`_LP64`	The data model where sizes of `long` and pointer are all 64 bits.

Note – The symbols `_ILP32` and `_LP64` 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 using `_ILP32` or `_LP64`. This makes the compilation environment machine independent and maximizes the portability of the application to all 64-bit platforms.

Compiler Environments

The Sun Studio C, C++, and Fortran compilation environments have been enhanced to support the creation of both 32-bit and 64-bit applications. The 10.0 release of the C compiler from Sun Studio provides 64–bit compilation support.

Native and cross-compilation modes are supported. The default compilation environment continues to produce 32-bit applications. While both modes are supported, they are still architecture-specific. It is not possible to create SPARC objects on x86 machines, nor x86 objects on SPARC machines with the Sun compilers. In the absence of a specification of the architecture or mode of compilation, the appropriate `__sparcv8` or `__i386` symbol is defined by default, and as part of this, `_ILP32` is also defined. This maximizes interoperability with the existing applications and hardware base.

Starting with the Sun Studio 8 release, use the `cc`(1) `-xarch=generic64` flag to enable the 64-bit compilation environment.

This generates LP64 code in ELF64 objects. ELF64 is a 64-bit object file format supporting 64-bit processors and architectures. This is in contrast to the ELF32 object files generated when compiling in the default 32-bit mode.

The `-xarch=generic64` flag is used to generate 64-bit code on either 32-bit or 64-bit system. Using the 32-bit compiler you can build 64-bit objects on a 32-bit system; however, you cannot run the 64-bit objects on a 32–bit system. You need not specify the library path for the 64-bit libraries. If the `-l` or `-L` option is used to specify an additional library or library path and that path points only to 32-bit libraries, the linker detects this and fails with an error.

For a description of compiler options, see the *Sun Studio 10: C User's Guide*.

32-bit and 64-bit Libraries

The 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 using 64-bit libraries. The 32-bit libraries continue to be located in `/usr/lib` and `/usr/ccs/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 using the `64` symbolic link.

In order to build 64-bit applications, you need 64-bit libraries. It is possible to do either native or cross-compilation, because the 64-bit libraries are available for both 32-bit and 64-bit environments. The compiler and other miscellaneous tools (for example; `ld`, `ar`, and `as`) are 32–bit programs capable of building 64-bit programs on 32-bit or 64-bit systems. Of course, a 64-bit program built on a system running the 32-bit operating system cannot execute in that 32-bit environment.

Linking Object Files

The linker remains a 32-bit application, but this should be transparent to most users, since it is normally invoked indirectly by the compiler driver, for example, `cc(1)`. 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.

`LD_LIBRARY_PATH` Environment Variable

SPARC. The two separate dynamic linker programs for 32-bit applications and for 64-bit applications are: `/usr/lib/ld.so.1` and `/usr/lib/sparcv9/ld.so.1`.

x86. For the AMD64 architecture, the dynamic linker programs for 32-bit applications and 64-bit applications are: `/usr/lib/ld.so.1` and `/usr/lib/amd64/ld.so.1`.

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 completely overridden using the `LD_LIBRARY_PATH_64` environment variable.

`$ORIGIN` Keyword

A common technique for distributing and managing applications is to place related applications and libraries into a simple 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 using NFS, Sun's distributed computing file system, and mounted on client machines. 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 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 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`.

See the *Linker and Libraries Guide* for further details.

Packaging 32-bit and 64-bit Applications

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

Placement of Libraries and Programs

SPARC. The placement of new libraries and programs follows the standard conventions described in "32-bit and 64-bit Libraries" on page 43. 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.

This means that 32-bit libraries should be placed in the same library directories. 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.

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.

See "Application Naming Conventions" on page 46.

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.

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 "Placement of Libraries and Programs" on page 45. 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.

Shell-Script Wrappers

In the case where 32-bit and 64-bit specific versions of applications are required, shell-script wrappers can make the version transparent to the user. This is the case with a number of tools in the Solaris operating environment, where 32-bit and 64-bit versions are needed. A wrapper can use the `isalist` command to determine the native instruction sets executable on a particular hardware platform, and run the appropriate version of the tool based on this.

This is an example of a native instruction set wrapper:

```
#! /bin/sh

CMD=‘basename $0‘
DIR=‘dirname $0‘
EXEC=
for isa in ‘/usr/bin/isalist‘; do
    if [-x ${DIR}/${isa}/${CMD}]; then
        EXEC=${DIR}/${isa}/${CMD}
        break
    fi
done
if [-z "${EXEC}"]; then
        echo 1>&2 "$0: no executable for this architecture"
        exit 1

fi
exec ${EXEC} "${@}"
```

One problem with this example is that it expects the `$0` argument to be a full pathname to its own executable. For this reason, a generic wrapper, `isaexec()`, has been created to address the problem of 32-bit and 64-bit specific applications. A description of this wrapper follows.

`/usr/lib/isaexec` Binary File

`isaexec(3C)` is a 32-bit executable binary file that performs the wrapper function outlined in the shell script wrapper presented in the immediately preceding description, but with precise preservation of the argument list. The executable's full pathname is `/usr/lib/isaexec`, but it is

not designed to be executed by that name. Rather, it can be copied or linked (hard link, not soft link) to the primary name of a program that exists in more than one version, selected using `isalist(1)`.

For example, in a SPARC environment, the command `truss(1)` exists as three executable files:

```
/usr/bin/truss
/usr/bin/sparcv7/truss
/usr/bin/sparcv9/truss
```

The executables in the `sparcv7` and `sparcv9` subdirectories are the real `truss(1)` executables, 32-bit and 64-bit respectively. The wrapper file, `/usr/bin/truss`, is a hard link to `/usr/lib/isaexec`.

In the x86 environment, the command `truss(1)` exists as three executable files:

```
/usr/bin/truss
/usr/bin/i86/truss
/usr/bin/amd64/truss
```

The `isaexec(3C)` wrapper determines its own fully resolved symlink-free path name using `getexecname(3C)`, independent of its `argv[0]` argument, gets the `isalist(1)` through `sysinfo(SI_ISALIST, ...)`, and performs an `exec(2)` of the first executable file bearing its own name found in the resulting list of subdirectories of its own directory. It then passes the argument vector and environment vector unchanged. In this way, `argv[0]` passed to the final program image appears as first specified, not as transformed into a full path name modified to contain the name of the subdirectory.

Note – Because wrappers might exist, you need to be careful when moving executables to different locations. You might move the wrapper rather than the actual program.

`isaexec(3c)` Interface

Many applications already use startup wrapper programs to set environment variables, clear temporary files, start daemons, and so on. The `isaexec(3C)` interface in `libc(3LIB)` allows the same algorithm used in the shell-based wrapper example above to be invoked directly from a custom wrapper program.

Debugging 64-bit Applications

All of the 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 Sun Studio tool suites. The remaining tools are included with the Solaris release.

The options for all these tools are unchanged. 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 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 64–bit environments.

SPARC V9 ABI Features

64-bit applications are described 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, contains details of the SPARC V9 ABI. It describes the 32-bit SPARC V8 ABI and the 64-bit SPARC V9 ABI. You can obtain this document from SPARC International at www.sparc.com.

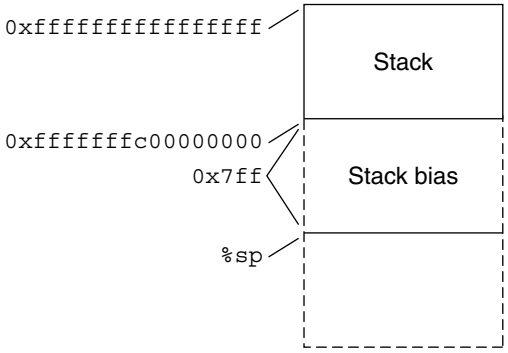
Following is a list of the SPARC V9 ABI features.

- 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. See *The SPARC Architecture Manual, Version 9*.
- 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. Results are returned in `%o0`. Because all registers are now treated as 64-bit quantities, 64-bit values can now be passed in a single register, rather than a register pair.
- The layout of the stack is different. Apart from the increase in the basic cell size from 32-bit to 64-bit, various hidden parameter words have been removed. 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 bits. 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).
- 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 SPARC applications and those for 64-bit SPARC applications.

Stack Bias

SPARCV9. 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.

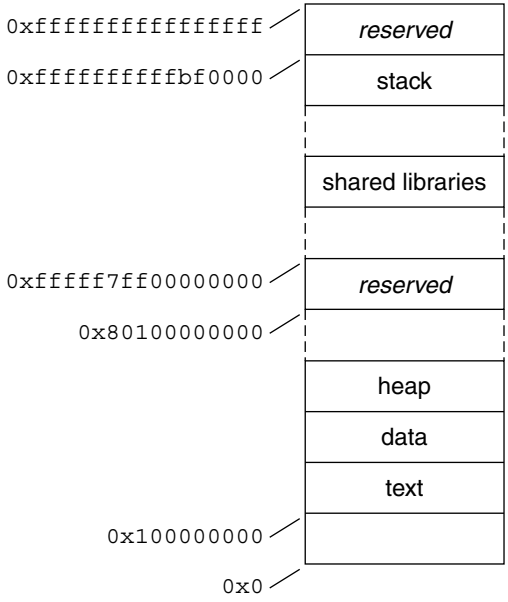


For more information on stack bias, please see the SPARC V9 ABI.

Address Space Layout of the SPARC V9 ABI

SPARCV9. 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 machine, and are given for illustrative purposes only.

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.

Code Models of the SPARC V9 ABI

SPARCV9. 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:

- Positionability (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 6–1 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
`pic`	PIC	< 2 gigabytes	Anywhere	Small (<= 1024 external objects)
`PIC`	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 `abs64` 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.

See the SPARC V9 ABI and compiler documentation for more information on code models.

Note – A program compiled with the `abs32` code model must be linked below 4 gigabytes using the `-M /usr/lib/ld/sparcv9/map.below4G` option.

AMD64 ABI Features

64-bit applications are described 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`).
  Two registers are temporaries (`%r10` and `%r11`).
  Five registers are callee-saved (`%r12`, `%r13`, `%r14`, `%r15`, and `%rbx`)
- 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 that order.
- 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.
- Instruction sizes are still 32 bits. 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.

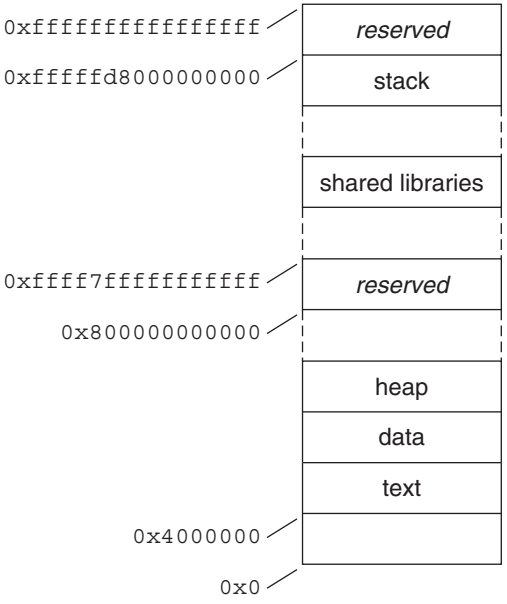
  The 64-bit ABI allows all the x87 and MMX instructions that operate on the x87 floating point registers (`%fpr0` through `%fpr7` and `%mm0` through `%mm7`) to be used. Additionally, the full set of SSE and SSE2 instructions that operate on the 128-bit XMM registers (`%xmm0` through `%xmm15`) can be used.

See the draft amd64 psABI document *System V Application Binary Interface, AMD64 Architecture Processor Supplement*, Draft Version 0.92.

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 systems.



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

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 6–2 Data Type Alignment

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

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(3DOOR)` 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 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 using compiler directives such as `#pragma pack` or `_Pack`. See "Alignment Issues" on page 55.

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.

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 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, Solaris provides a set of GELF (Generic ELF) interfaces that allow the programmer to manipulate both formats using a single, common API. See `elf(3ELF)`.

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

/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.

Extensions to sysinfo(2)

New sysinfo(2) subcodes in the Solaris S10 operating environment 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 using the SI_ARCHITECTURE_64 subcode. See sysinfo(2) for details.

libkvm and /dev/ksyms

The 64-bit version of the Solaris system is implemented using a 64-bit kernel. 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 Solaris platform, to extract the needed data with system calls. See sysinfo(2), kstat(3KSTAT), sysconf(3C), and proc(4) as the most common alternative APIs. If these interfaces can be used instead of kvm_open(3KVM), use them and leave the application as 32-bit 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.

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 (see kstat(3KSTAT)) 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.

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.

Changes to stdio

In the 64–bit environment, the stdio facility has been extended to allow more than 256 streams to be open simultaneously. The 32–bit stdio facility continues to have a 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. See, for example, __fbufsize(3C).

Performance Issues

The following sections discuss advantages and disadvantages of 64–bit performance.

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.

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.
- 64–bit applications do not run on 32–bit platforms.

System Call Issues

The following sections discuss system call issues.

What Does `EOVERFLOW` Mean?

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.

Beware `ioctl()`

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:

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

Both `ioctl(2)` calls work correctly when this code fragment is compiled and run as part of a 32-bit application.

Both `ioctl()` calls also return success when this code fragment is compiled and run as a 64-bit application. However, neither `ioctl()` works correctly. The first `ioctl()` 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()` 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 default 32-bit compilation environment is identical to historical Solaris operating environment 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.

Notice 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 better understand the options available to the applications developer, it helps to understand the `_ILP32` and `_LP64` feature test macros.

TABLE A–1 Feature Test Macros

Feature Test Macro	Description
`_ILP32`	The `_ILP32` feature test macro is used to specify the ILP32 data model where `int`s, `long`s and pointers are 32-bit quantities. By itself, the use of this macro makes visible those derived types and sizes identical to historical 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`	The `_LP64` feature test macro is used to specify the _LP64 data model where `int`s are 32 bit quantities and `long`s and pointers are 64 bit quantities. _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.

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

EXAMPLE A–1 `size_t` Defined in `_LP64`

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

EXAMPLE A–1 `size_t` Defined in `_LP64` *(Continued)*

```
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 quantity in the LP64 model. In contrast, when building a 32-bit application, `size_t` is defined as an `uint_t`, or `unsigned int`, a 32-bit quantity in either in the ILP32 or the LP64 models.

EXAMPLE A–2 `uid_t` Defined in `_LP64`

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

In either of these examples, the same end 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 a `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 Solaris operating environment.

Table A–2 lists the derived types that have changed. Notice that the types listed under the `_ILP32` feature test macro match those in Solaris 2.6, before 64–bit support was added to the Solaris software. 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, `ints`, `longs`, and pointers are 32-bit quantities. In the 64-bit environment, `ints` are 32-bit quantities while `longs` and pointers are 64-bit quantities.

TABLE A–2 Changed Derived Types — General

Derived Types	Solaris 2.6	_ILP32	_LP64
`blkcnt_t`	`longlong_t`	`longlong_t`	`long`
`id_t`	`long`	`long`	`int`
`major_t`	`ulong_t`	`ulong_t`	`uint_t`
`minor_t`	`ulong_t`	`ulong_t`	`uint_t`
`mode_t`	`ulong_t`	`ulong_t`	`uint_t`

TABLE A–2 Changed Derived Types — General *(Continued)*

Derived Types	Solaris 2.6	_ILP32	_LP64
nlink_t	ulong_t	ulong_t	uint_t
paddr_t	ulong_t	ulong_t	*not defined*
pid_t	long	long	int
ptrdiff_t	int	int	long
size_t	uint_t	uint_t	ulong_t
ssize_t	int	int	long
uid_t	long	long	int
wchar_t	long	long	int
wint_t	long	long	int

Table A–3 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. Notice that the ILP32 compilation environment has been preserved with the previous Solaris 2.6 release.

TABLE A–3 Changed Derived Types — Large File Specific

Derived Types	Solaris 2.6	_ILP32	_LP64
blkcnt64_t	longlong_t	longlong_t	blkcnt_t
fsblkcnt64_t	u_longlong_t	u_longlong_t	blkcnt_t
fsfilcnt64_t	u_longlong_t	u_longlong_t	fsfilcnt_t
ino64_t	u_longlong_t	u_longlong_t	ino_t
off64_t	longlong_t	longlong_t	off_t

Table A–4 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, then _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 A–4 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

TABLE A–4 Changed Derived Types — `FILE_OFFSET_BITS` Value *(Continued)*

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 (FAQs)

How can I tell if my system is running the 32-bit or the 64-bit version of the operating system?

You can determine what applications the operating system can run using the `isainfo -v` command. It displays the set of applications supported by the operating system. See the `isainfo(1)` man page for more information.

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 a 32-bit operating system?

No. Your application does not require changes or recompilation if it is being executed only on a system running the 32-bit operating system.

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 a system running the 64-bit operating system without requiring code changes or recompilation. Those 32-bit applications not requiring 64-bit capabilities can remain 32-bit to maximize portability.

If your application uses `libkvm(3LIB)` , it must be recompiled as 64-bit, to execute on a system running the 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 quantities involved.

What program do I need to invoke in order to get the 64-bit capabilities?

No program is available that specifically invokes 64-bit capabilities. In order to take advantage of the 64-bit capabilities of your system running the 64-bit version of the operating system, you need to rebuild your application.

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

Yes. Both native and cross-compilation modes are supported. The default compilation mode is 32-bit, whether on a system running the 32-bit or 64-bit version of the operating system.

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

Yes, provided you have the system headers and 64–bit libraries installed. However, it is not possible to run the 64-bit application on a system running the 32-bit operating system.

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 4–1.

What about `time_t`?

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

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

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

Can I create 64–bit XView or OLIT Applications?

No. These libraries are already obsolete for the 32–bit environment and will not be carried forward to the 64–bit environment.

Why is there a 64–bit version of `ls` in `/usr/bin/sparcv9/ls`?

In normal operation, there is no need for a 64–bit version of `ls`. However, since it is possible to create file system objects in `/tmp` and `/proc` that are “too large” for 32–bit `ls` to understand, the 64–bit version of `ls` allows users to examine those objects.

Index