FUJITSU COMPUTERS SIEMENS

Reliant UNIX ONLINE Documentation

Reliant UNIX 5.44

Porting Guide

Edition October 1997

Identification: U25614-J-Z915-1-7600

Copyright and Trademarks

1 Introduction

The Reliant UNIX 5.44 release is a follow-on product to Reliant UNIX 5.43c and is available on all SNI platforms based on R4000 and R10000 processors. The functionality of the **64-bit pointer** is introduced within the scope of Reliant UNIX 5.44 using the LP64 data model. This model allows applications to access large virtual and physical address spaces. Applications that need this functionality must be adapted to facilitate the use of 64-bit pointers.

The 64-bit environment is implemented in addition to a 32-bit runtime and generation environment. As usual, Reliant UNIX 5.44 is fully backwards compatible with earlier Reliant UNIX releases. This means that 32-bit Userland applications can still be run successfully.

Applications can not only access large memory, but can also use files that are larger than 2 Gbytes. Reliant UNIX contains an implementation of the **LFS** interfaces *(Large File Summit)*, which even allow 32-bit applications to access large files. This guide describes how applications can use these interfaces.

The manual contains guidelines and information on adapting application programs and kernel modules for accessing the new functions of Reliant UNIX 5.44. It is divided into the following sections:

Porting to 64-bit

This chapter describes how an existing application is ported to the 64-bit environment and can thus access large virtual memory. This chapter is intended for users who want to port applications to 64-bit.

Using the LFS interfaces

Certain software products may still be available as 32-bit applications because they do not need to use virtual memory over the 2-Gbyte limit. On the other hand, however, it may be necessary for these applications to access large files. This chapter is of interest to developers who want to access large files from their applications.

System call interface

This chapter contains detailed information about the system call interface.

Kernel-specific guidelines

This chapter is intended for developers who want to port a kernel component to the 64-bit environment. It describes strategies for converting data between the 64-bit kernel and the 32-bit user application, as well as kernel-specific information, including DDI/DKI changes.

1.1 Target group

This guide is intended for all developers who want to port their products to the 64-bit environment or LFS on Reliant UNIX systems. It covers all aspects of our development that are relevant for such ported software, including user program development, *libc*/kernel interfaces and interfaces within the kernel.

1.2 Notational conventions

The following notational conventions are used in this manual:

Italics	in the main body of text denote file names, program names, commands, options, constants and variable names
Typewriter text	indicates system output, such as error messages, messages or file excerpts
"Quotation marks"	denote references to other chapters or manuals and highlighted entries in continuous text

 Additional information, notes and tips

 Warnings that must be heeded□

2 Porting to 64-bit

This chapter contains general information about porting an existing application to the new 64-bit environment of Reliant UNIX 5.44.

2.1 Supported interfaces

The changes that are necessary for a 64-bit operating system extend over the entire kernel and can also affect user programs. Every effort was made during development to ensure the continued compatibility with the 32-bit Userland programs while making changes to the 64-bit operating system. Compatibility in this context involves two aspects:

- It must still be possible to run existing 32-bit binary programs (binaries) that were ported to earlier Reliant UNIX releases and that do not have any specific dependencies on the 64-bit functionality of the new operating system. Such programs normally use the MIPS1 or MIPS2 instruction set.
- The source code must still be compatible; this means that SVID, POSIX and XPG4 interfaces, as well as all existing Reliant UNIX 32-bit APIs are still available.

In addition, a **64-bit generation and runtime environment** which uses the MIPS3 instruction set (for R4000) and the MIPS4 instruction set (for R10000) is defined. The runtime environment uses a new object code format, the **64-bit Small Code Model** (64s).

The following diagram shows the structure of the 32-bit and 64-bit Reliant UNIX 5.44 environments:

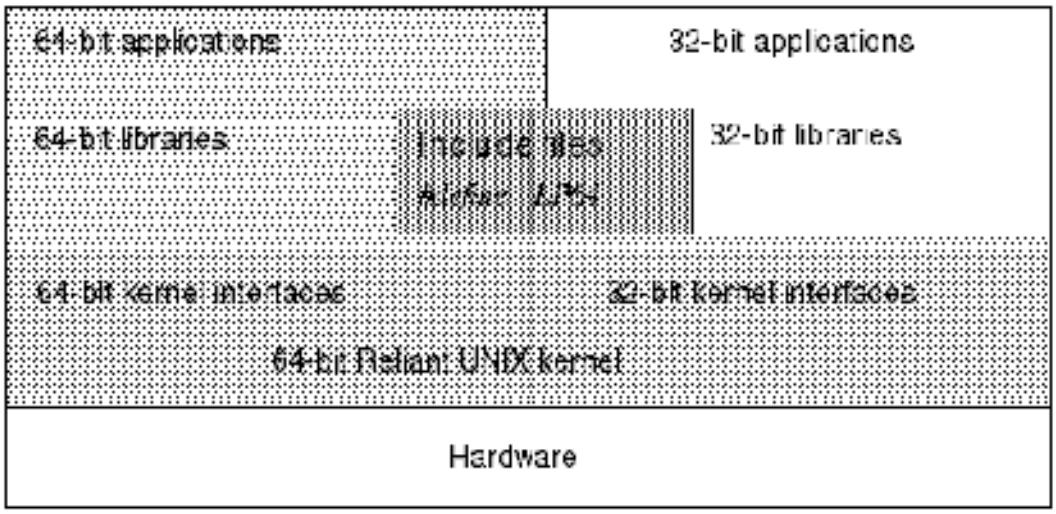

Figure 1: Structure of the 32-bit and 64-bit Reliant Unix 5.44 environments

Essentially the diagram shows the following features of Reliant UNIX 5.44:

- The kernel is a pure 64-bit application. This means that each kernel component (driver, additional file system, Streams module, etc.) must be ported to 64-bit (see below). The id commands, in particular, e.g. *idbuild*, can only be used for generating 64-bit kernels. Reliant UNIX 5.44 does not support the generation of 32-bit kernels or the use of 32-bit kernel components.
- The kernel provides interfaces for 32-bit and 64-bit applications. 32-bit data structures are converted to the 64-bit view of a data structure within the kernel at the system call level (syscall).
- There are two sets of libraries: a set of 64-bit libraries and a set of 32-bit libraries for 64-bit and 32-bit applications respectively. The 32-bit libraries (Dynamic Shared Objects, DSOs) and the conversion function of the kernel together form the 32-bit system runtime environment. The 64-bit run-time environment is implemented using the 64-bit DSOs with the 64-bit interfaces to the kernel.

  Some libraries do not as yet have a 64-bit version. The section entitled Section "List of 64-bit libraries" contains a complete list of the 64-bit libraries and their memory addresses within the system.

- The include files can be used for generating both 32-bit and 64-bit applications. The define statement *__LP64__* is specified to provide the application with the 64-bit view of the data structures. This statement is normally set by the compiler (see Section "Compiler options"). The generation of the 64-bit environment is defined by specifying the *#define* statement *__LP64__* and using the 64-bit libraries. The 32-bit environment is implemented by resetting the *__LP64__* setting and using the 32-bit libraries.
- Objects, DSOs and libraries from different environments (32-bit or 64-bit) **cannot** be connected to an executable program using links.
- Only **one** command set is provided. Almost all the commands are 32-bit applications. Only the commands that relate to the kernel memory allocation or large data objects are ported to 64-bit (see Section "List of the 64-bit commands").

2.2 The LP64 data model

The compiler uses the LP64 model during compilation. This specifies that the length of long data and pointers is 64 bits. However, the 32-bit length is still retained for integers. The following table shows the size and alignment of the most important basic types in bytes.

	32-bit model		**LP64 model**	
	Size	Alignment	Size	Alignment
char	1	1	1	1
short	2	2	2	2
int	4	4	4	4
long long	8	8	8	8
long	4	4	8	8
float	4	4	4	4
double	8	8	8	8
pointers	4	4	8	8

Table 1: Basic types of the supported data models

2.3 Include files

A check was carried out on all include files to ensure that the structure alignments and type definitions in the LP64 model were still valid. If this was not the case, the relevant changes were then made. In order to maintain the source compatibility for 32-bit applications, each change was embedded in *#ifdef __LP64__*. If the type definitions were changed, every effort was made to guarantee source compatibility with third-party 64-bit operating systems. New integer types were also included in the *sys/types.h* file for the same reason.

Type	32-bit mode	64-bit mode
__int32_t	int	int
__uint32_t	unsigned int	unsigned int
__long32_t	long	int
__ulong32_t	unsigned long	unsigned int
__long64_t	long long	long
__ulong64_t	unsigned long long	unsigned long

Table 2: New integer data types

The *-Kl64 -D_LONGLONG* options must be used for the compiler for accessing *__long64_t* and *__ulong64_t* in 32-bit mode.

2.4 List of 64-bit libraries

The following libraries are available in the 32-bit and 64-bit version. The 64-bit directory for the library is listed in the "Directory" column in the table below. The 32-bit versions of the specified libraries are stored in the same location as in previous Reliant UNIX versions:

Library	Directory
libext.a	/usr/lib64s
libmproc.so	/usr/lib64s
libsocket.so	/usr/lib64s
npack.so	/usr/lib64s
libext.so	/usr/lib64s
libns.a	/usr/lib64s
libspecaio.a	/usr/lib64s
resolv.so	/usr/lib64s
libbsocket.so	/usr/lib64s
libio.a	/usr/lib64s
libnsl.so	/usr/lib64s
libxnet.so	/usr/lib64s
straddr.solibc.so.1	/usr/lib64s
libio.so	/usr/lib64s
libnsl_i.so	/usr/lib64s
libxnet.so.1	/usr/lib64s
tcpip.so	/usr/lib64s

libdl.so	/usr/lib64s
libmproc.a	/usr/lib64s
libresolv.so	/usr/lib64s
libcurses.a	/usr/lib64s
libnsl.a	/usr/lib64s/netlib
libstraddr.a	/usr/lib64s/netlib
straddr.a	/usr/lib64s/netlib
libsocket.a	/usr/lib64s/netlib
libtcpip.a	/usr/lib64s/netlib
tcpip.a	/usr/lib64s/netlib
libc.a	/usr/ccs/lib64s
libc.so	/usr/ccs/lib64s
libgen.a	/usr/ccs/lib64s
libm.a	/usr/ccs/lib64s
64-bit startup files	/usr/ccs/lib64s

Table 3: List of the 64-bit libraries

2.5 List of the 64-bit commands

Some of the Reliant UNIX 5.44 commands were ported to 64-bit in accordance with the modified kernel structures of the 64-bit data model.

Other commands were ported on the basis of the LFS specifications (in compliance with XPG5).

The following commands relate to 64-bit applications.

There is no 32-bit version on the system for these commands:

cat, *cd*, *chgrp*, *chmod*, *chown*, *cksum*, *cmp*, *cp*, *dd*, *df*, *dirname*, *du*, *dumpsave*, *find*, *ln*, *ls*, *mkdir*, *mv*, *pathchk*, *pwd*, *rm*, *rmdir*, *sum*, *test*, *touch*, *ulimit*, *compress*, *uncompress*, *tar*, *cpio*, *ksh*, *csh*, *sh*, *crash*, *dkstat*, *fusage*, *fuser*, *gated*, *idload*, *ifadmin*, *iotail*, *ipcs*, *pfprivs*, *prfld*, *proceso*, *sar*, *setikdb*, smtpd, *statis*, *swap*, *sysdef*, *trtpt*, *truss*, *uname*, *dssi_dd*, *panel*, *quickdd*.

All of these commands are still compatible with the corresponding earlier versions from the point of view of options and Exit codes.

Although the generation environment commands, e.g. *cc*, *ld*, *debug* etc. also still comprise 32-bit binaries, they have been changed in such a way that they can also generate 64-bit objects.

2.6 Compiler options

You can use the *cc* command to access the correct environment in the following way:

When you type the *-Klp64* option in the command line of the *cc* command, the *__LP64__ #define* statement and the LP64 data model are selected during compilation, the 64-bit libraries and the start files are linked and the compiler generates a binary program that corresponds to the alignment of the 64-bit Small Code model.

You can use the CDS++ 1.0 and pyrc 6.0 compilers, with debugger, for generating your programs.

If you do not specify the *-Klp64* option, the compiler generates a 32-bit binary program using the 32-bit versions of the libraries.

2.7 32-bit applications on LP64

2.7.1 General information on porting

When porting applications to the 64-bit environment, problems can arise whenever assumptions are made on the following points:

- The length of the types int, long and pointer
- Correspondence between the length of the types int, long and pointer
- The importance of the alignment of a structure (e.g. for reading binary data from files)

This list only mentions some of the areas in which problems can arise. Other aspects that are important for porting are described in more detail in later chapters.

In addition, any components of applications that have hardware dependencies or that read the kernel memory must also be adapted (see below).

2.7.2 Type conversion

The use of all int and long data types must be checked when converting a 32-bit code to a 64-bit code.

In the case of the int type, it is important to check whether the variable (or the data field) should be scaled to 64 bits, or should be left at 32 bits for reasons of compatibility. If the data field is to be scaled to 64 bits, the type must be changed to long; otherwise the int type should not be changed. Since the use of the long type doubles the length of these data fields, a conversion from int to long should only be performed if it is necessary for technical reasons.

The same rules apply in the opposite direction for the long type: Does the data field in question have to remain a 32-bit field for reasons of compatibility or can it be extended to 64 bits? If the size is to remain at 32 bits, the data field should be changed to the int type.

Furthermore, each use of int should be checked when explicit type casting is carried out. In certain instances, it may no longer be possible to force a variable into the int type.

Type casting of the long type in arithmetic expressions must also be checked since such conversions can produce unexpected results if signs are added or changed.

Changing int to long (or vice versa) in include files normally results in incompatibility. If you want to change a type in include files, you must therefore embed the type in the *#ifdef__LP64__* statement.

2.7.3 Format strings

In 32-bit environments, int (Integer) and long (long Integer) data types are the same size. However, the long and pointer data types are different from int data types.

This is not always immediately apparent in formatted output statements. Use the format statements *%lx* and *%ld* for long or pointer data types.

The *lint* program generates warnings whenever there are discrepancies between format strings and arguments. The warning function is also included in the compiler.

Pointers should therefore be output as follows:

printf("%lx\n", (long)ptr);

and variables of the type long should be output in decimal format as:

printf("%ld\n", long_var);

pointer and long data types that are output in this way are displayed correctly in both 32-bit and 64-bit mode.

2.7.4 Pointer arithmetic

With pointer arithmetic, all allocations of pointers to integers must be changed to allocations of pointers to long. This is often the case with page alignment. The next example shows a fail-safe method of aligning a pointer. This method works perfectly in both 32-bit and 64-bit programs:

```
#include <unistd.h>
int pagesize;
```

```
pagesize=sysconf(_SC_PAGESIZE);
ptr = (ptr_type *)(((long)ptr + pagesize - 1) & ~(pagesize - 1));
```

2.7.5 Interprocess communication

The following communication mechanisms can be used between 32-bit and 64-bit applications provided **no** assumptions are made with regard to the alignment of binary data between the processes:

- Network communication

  The network file system NFS can be used on 32-bit and 64-bit systems without restrictions for file sizes less than 2 Gbytes. If you want to exchange large files over NFS, you must have NFS V3.0 installed on both the client system and the server system. NFS V3.0 is available under Reliant UNIX version 5.43a and higher.

  Two processes can communicate over RPC and XDR.

  Socket connections can be used provided no assumptions are made regarding the alignment of data (see next example).

- Interaction using system calls

  32-bit and 64-bit processes can communicate over pipes, shared memory and shared file access, subject to the restrictions mentioned above.

Example

Abtypes.h contains the following declaration of *struct c*:

```
struct c {
:
:
void *ptr;
:
long long_var;
:
}
```

This is based on the following C sources *a.c* and *b.c*:

a.c:	b.c:
#include Abtypes.h□ {□ struct c d;□ :□ :□ write(fd, d, sizeof(d));□ :□ :□ }	#include Abtypes.h□ {□ struct c *d;□ char buf[BUFLEN];□ :□ :□ read(fd, buf, BUFLEN);□ d = &buf;□ :□ :□ }

If you compile *a.c* and *b.c* in different bit environments, the binaries that are generated from this cannot exchange data over *struct c* because *struct c* has different alignments in the 32-bit and 64-bit environment.

2.7.6 Hardware and kernel dependencies

The kernel becomes a pure 64-bit program, but both 32-bit and 64-bit programs can be run. All kernel modules must be recompiled as 64-bit object files (see below). The alignment and field sizes must be checked when porting to the LP64 data model in order to satisfy all hardware requirements. The size and/or alignment of structures that are used to communicate between the kernel and user programs may change.

In the case of programs that work with memory sizes or addresses, it is important to note that the kernel pointers and many memory sizes do not fit in the 32-bit format. All programs that use kernel addresses or other program addresses, or memory sizes should be converted to 64-bit programs. For example, programs that use the following interfaces should be ported to 64-bit:

- Programs that read */dev/kmem*
- Programs that call *statis*, *ptrace*
- Programs that use the *ioctl* calls PIOCGETPR (read struct proc) and PIOCGETU (read struct user)

If the program you use for reading the */dev/kmem* kernel memory is to remain a 32-bit program, the following preconditions must be satisfied:

- The 32-bit program and the 64-bit kernel need to have the same view of the data structures to be read. This means that these data structures cannot contain any of the following basic data types: long, unsigned long, long long, unsigned long long and pointers to any data types. (If a program which reads kernel memory needs these data types, the program must be ported to 64-bit).
- 64-bit wide offsets must be used for searching in the */dev/kmem* file. This can be achieved in a 32-bit program by using the data type long long for the offset and the LFS system call *lseek64* for the search in */dev/kmem*. However, the file must first be opened using the appropriate 64-bit Open flag in order to allow use of *lseek64* system calls.

The following condition must be satisfied if *statis* is to remain a 32-bit program:

- The 32-bit program and the 64-bit kernel need to have the same view of the data structures to be read. This means that these data structures cannot contain any of the following basic data types: long, unsigned long, long long, unsigned long long and pointers to any data types. (If a *statis* program needs to access these data types, the program must be ported to 64-bit).

Programs that call *ptrace* and PIOCGETPR cannot remain as 32-bit programs (see above).

Programs that are used to read kernel memory and comply with these porting guidelines can be compiled in 64-bit mode. The field width in *printf* statements may have to be checked in order to ensure that the display of "larger" values is still readable.

If you change a program used for reading the kernel memory, you should also convert the program to *statis* if

at all possible rather than reading */dev/kmem* directly. (Writing to the kernel memory is not supported by *statis*, with the exception of clearing statistical counters). We recommend that you specify most fields in statistics structures as unsigned integers. This should not create any problems for counter statistics. The long type must be used for statistics structures that provide information on memory sizes, e.g. *kmeminfo*, so that larger memory sizes can also be included.

2.8 Memory allocation addresses

The standard memory allocation addresses for shared memory, mapped files and shared libraries change in 64-bit applications. The memory allocation address for 32-bit programs does not change. The default text and data addresses also change along with the memory allocation address. The following table shows an overview of the changes.

	32-bit	**64-bit**
Text	0x40 0000	0x40 0000
Data	after the end of the text	after the end of the text
Stack	0x7fff 0000	0xff ffff 0000
Shared Libraries	≥ 0x0800 0000 and□ ≤ 0x08ff ffff	≥ 0x0800 0000 and□ ≤ 0x08ff ffff
Mapped Files	> 0x0800 0000 and□ ≤ 0x08ff ffff and□ > Shared Libraries	> 0x0800 0000 and□ ≤ 0x08ff ffff and□ > Shared Libraries
Shared Memory	see Mapped Files	see Mapped Files

Table 4: Memory allocation addresses

2.9 Determining the runtime environment

If you include the *net/netstatis.h* file, you can access the *int is_64bit_machine()* routine which generates a non-null value when it is called on a 64-bit machine. You can also use the following *nstatis* system call:

nstatis(0,"netstatis_config",STATIS_GET,result,sizeof(int))

where result is an int pointer (see man page *nstatis*). If result & NETSTATIS_LP64 are not null, your application will run on a 64-bit machine.

3 Using the LFS interfaces

This chapter describes the various options for accessing large files within □
Reliant UNIX 5.44. The information in this chapter is only relevant for 32-bit applications, since large files are the norm for 64-bit applications. If you are not yet familiar with the LFS specification, you can find out more about it at the following URL address:

http://www.sas.com/standards/large.file/x_open.20Mar96.html

3.1 off_t for 32-bit binaries

A number of different methods can be used to define the size of off_t. To enable the programs to use different environments, they must be compiled for each specific environment. If the programs are to use the functions described in this section, they must be compiled using the new Compiler and Linker options. The *getconf* utility can be called with the new arguments in order to generate the Compiler and Link options.

Since the various sizes of off_t are defined in the unistd.h file, you must make sure that unistd.h is the first include file in your source.

Method 1:

Compile a program with additional APIs and a selectable size for off_t. This program can use *fseeko()* and *ftello()*. The off_t environment is "large" (64-bit).

```
$(CC) -D_LARGEFILE_SOURCE  \□
$(getconf LFS_CFLAGS)  file.c  \□
$(getconf LFS_LDFLAGS)  \□
$(getconf LFS_LIBS)
```

Method 2:

Compile a program with a selectable size for off_t. This program does not use *fseeko()* and *ftello()*. The off_t environment is "large" (64-bit).

```
$(CC) $(getconf LFS_CFLAGS)  file.c  \□
$(getconf LFS_LDFLAGS)  \□
$(getconf LFS_LIBS)
```

Method 3:

Compile a program with additional APIs and a default size for off_t. This program can use *fseeko()* and *ftello()*. The off_t environment is "small" (32-bit).

```
$(CC) -D_LARGEFILE_SOURCE   file.c
```

Method 4:

Compile a program with a transition interface or an explicit API. This program uses versions of SUS interfaces (Single UNIX Specification) such as *lseek64()* and *fopen64()*.

```
$(CC) -D_LARGEFILE64_SOURCE  \□
$(getconf LFS64_CFLAGS)  file.c  \□
$(getconf LFS64_LDFLAGS)  \□
$(getconf LFS64_LIBS)
```

Our implementation allows object files compiled using different off_t environments to be linked. For example, an object module compiled in a 32-bit off_t environment can be linked with an object module compiled in a 64-bit off_t □
environment. In this case, API calls for 32-bit off_t and 64-bit off_t can be used in the same file descriptor. Refer to the LFS specification for more detailed information on linking.

The following LFS FLAGS are used in this guide to illustrate this. (**Note:** These flags are only used here in order to explain linking and can be changed in the future. Given this, you should use the *getconf* method described above in make files and shell scripts):

```
LFS_CFLAGS:    "-D_FILE_OFFSET_BITS=64 -D_LONGLONG -Kll64"□
LFS_LDFLAGS:   ""□
LFS_LIBS:      ""□
LFS64_CFLAGS:  "-D_LONGLONG -Kll64"□
LFS64_LDFLAGS: ""□
LFS64_LIBS:    ""
```

3.2 Outputting off_t values

If you use 64-bit off_t values in your application, you must change the format strings of all statements of the *printf* family for outputting values of the type off_t in such a way that the ll format is used (ll stands for long long, and long long is used to implement the 64-bit off_t type).

Example

The statement

```
printf("This is off_t:  %d", off_t_var);
```

must be changed to

```
printf("This is off_t:  %lld", off_t_var);
```

4 System call interface

4.1 Calling convention

The number of system calls for 32-bit and 64-bit applications is still the same. The differentiation between 32-bit and 64-bit system calls is made within the kernel and is not therefore apparent in applications. The 64-bit version of the C library uses 8 registers to forward system call arguments to the kernel. Likewise, in 32-bit mode, the *64 system calls of the LFS interface (*lseek64*, *mmap64,* etc.) use 8 registers. The registers to be used are a0-a3 and t0-t3.

The system call interface copies all the arguments for the system calls into an array of data of the type long. The arguments come either from the registers for 64-bit arguments or from arguments that exist in the stack. Copying the arguments into the array of the type long changes these into long data types. The long array is then passed on to the individual system calls which adapt it to the "system call arguments" structure. The structures for "system call arguments" can only comprise fields of the type long.

In 32-bit programs, a sign extension to 64 bits already exists for register arguments. The remaining arguments are read as integers from the user stack and a sign is added before they are written to the array of system call arguments.

This convention allows the same argument structure for system calls to be used both for 32-bit and 64-bit programs.

Our objective would be to also apply the same argument structure for system calls for the *64 system calls of the LFS interface (*lseek64*, *mmap64,* etc.). To do this, we need a system call interface for the 32-bit kernel which is compatible with the 64-bit kernel. We have implemented an interface in which all the arguments for the *64 routines are extended to 64 bits, and the upper and lower sections of this 64-bit argument are written as pairs of 32-bit registers. In addition, the system call should use the eight argument registers a0-a3, t0-t3. The stack must be adapted according to the size of the 4 long long arguments. In other words, the fifth argument (4 arguments in a0 - a3, t0 - t3) must be located in the memory stack + 4 * (sizeof(long long)).

The following calling conventions are used on the system call interface:

64-bit object	Arguments are read from the registers a0-a3, t0-t3.
***64 routine (32-bit binary)**	All arguments are converted to 64-bit types. The even registers contain the top section of a 64-bit argument and the uneven registers contain the bottom section. If there are more than 4 arguments (8 registers), the remaining arguments are read as 64-bit long data from the user stack. In integer arguments, the high word is already initialized to null.
Standard 32-bit system call	The arguments 1-4 are called from a0-a3, the other arguments are read as 32-bit sets from the user stack and are increased to 64 bits using a sign extension.

Table 5: Calling conventions

5 Kernel-specific guidelines

5.1 Changes to the kernel segment

The names and sizes of the kernel and user segments have been changed. The macros used in the kernel for accessing the virtual kernel memory and the physical main memory have been adapted. These segments were not supposed to be accessed directly in drivers. The following table lists the old macros and shows how they are to be replaced for kernel modules that use these segments.

Old macro	New macro
IS_KSEG0□ IS_KSEG1□ IS_KSEG2	IS_CACHED□ IS_UNCACHED□ IS_MAP
K0_TO_K1□ K1_TO_K0	CACHED_TO_UNCACHED□ UNCACHED_TO_CACHED
PHYS_TO_K0□ PHYS_TO_K1□ K0_TO_PHYS□ K1_TO_PHYS	PHYS_TO_CACHED□ PHYS_TO_UNCACHED□ CACHED_TO_PHYS□ UNCACHED_TO_PHYS

Table 6: Changed macros

5.2 ioctl/statis conversion

When porting *ioctls*, kernel users must decide whether they want to support both 32-bit and 64-bit access. The driver must be changed even when only 64-bit access is supported in order to ensure that 32-bit access to the driver is rejected.

```
#include <sys/types.h>□
□
switch (syscallcc())□
{□
case SYSCALL64:□
/* pure 64 bit executable */□
case SYSCALL32:□
/* pure 32 bit system call */□
case SYSCALL64_ON_32:□
/* *64 system call [lseek64, mmap64, etc in 32 bit binary */□
}
```

The following must be added in order to reject 32-bit access:

```
if (syscallcc() != SYSCALL64)□
return;
```

5.3 copyin and copyout

If the 64-bit kernel component is used exclusively by 64-bit Userland programs, a *copyin* / *copyout* conversion may not be required.

If 32-bit Userland programs also use the interface provided by the 64-bit kernel component, and you only have to copy structures that do not contain either long or pointer data types, no *copyin* / *copyout* conversions are required. Such structures have the same alignment within the 64-bit kernel and the 32-bit Userland program.

If 32-bit Userland programs are also to use the interface of the 64-bit kernel component, certain *copyin* / *copyout* conversions must be carried out between the 32-bit Userland programs and your kernel component. The following *sys/compat32.h* macros have been added for performing this conversion between a 32-bit Userland kernel and the 64-bit kernel:

copyin:	**Copy from the user area, i.e. convert to the kernel view**
COPYIN_ERR(uaddr, kaddr, structurname, errorno);	
COPYIN_RET(uaddr, kaddr, structurname, errorno);	
COPYIN_BRK(uaddr, kaddr, structurname, errorno);	
NCOPYIN_ERR(uaddr, kaddr, structurname, count, errorno);	
NCOPYIN_RET(uaddr, kaddr, structurname, count, errorno);	
COPYIN_ELEM_ERR(uaddr, kaddr, structurname, element, errorno);	
COPYIN_ELEM_RET(uaddr, kaddr, structurname, element, errorno);	
COPYIN_ELEM_BRK(uaddr, kaddr, structurname, element, errorno);	
COPYIN_LP(uaddr, kaddr)	

Table 7: copyin conversions

copyout:	**Convert to the 32-bit view, i.e. copy to the user area**
COPYOUT_ERR(kaddr, uaddr, structurname, errorno);	
COPYOUT_RET(kaddr, uaddr, structurname, errorno);	
COPYOUT_BRK(kaddr, uaddr, structurname, errorno);	
NCOPYOUT_ERR(kaddr, uaddr, structurname, count, errorno);	
NCOPYOUT_RET(kaddr, uaddr, structurname, count, errorno);	
COPYOUT_ELEM_ERR(kaddr, uaddr, structurname, element, errorno);	
COPYOUT_ELEM_RET(kaddr, uaddr, structurname, element, errorno);	
COPYOUT_ELEM_BRK(kaddr, uaddr, structurname, element, errorno);	
COPYOUT_LP(kaddr, uaddr)	

Table 8: copyout conversions

Example

Let's suppose we have the following structure declaration and the following code fragment of a 32-bit kernel:

```
struct example{
long    e_long;
int     e_int;
char    *e_ptr;
};...if (copyin((caddr_t)user_buf, (caddr_t)kernel_buf, sizeof(struct example)) == -1)
error = EFAULT;...if (copyout((caddr_t)kernel_buf, (caddr_t)user_buf, sizeof(struct example)) == -1)
error = EFAULT;
```

Using the macros declared in *sys/compat32.h*, the code fragment shown above could be compiled easily to code according to the following procedure which converts 32-bit Userland structures to 64-bit kernel structures (where kernel_buf has the type struct example *):

1. #include <sys/compat32.h>
2. Add the language construct *#pragma compat32* to your code to get the 32-bit view of the structure within your 64-bit kernel program. The *#pragma compat32* construct is a special extension of the C preprocessor. The definition of the 32-bit view of the data structure is used by the following code of the COPY* macro:

   ```
   #if defined(__LP64__)
   #pragma compat32 example example_32
   #endif
   ```

3. Replace the 32-bit code in the above example with

   COPYIN_ERR(user_buf, kernel_buf, struct example, EFAULT);□
   ...□
   COPYOUT_ERR(kernel_buf, user_buf, struct example, EFAULT);

The following code fragment shows the contents of a COPYOUT_ERR macro.

```
if (syscallcc() == SYSCALL32) {□
struct example_32 ex_32;□
(void) __copybyname(ex_32, *(kernel_buf));□
if (copyout((caddr_t)&ex_32, (caddr_t)user_buf, □
sizeof(struct example_32)) □
== -1) □
error = errorno;□
} else if (copyout((caddr_t)kernel_buf, (caddr_t)user_buf, □
sizeof(sizeof(struct example))) == -1)□
error = errorno;
```

__copybyname is an internal compiler function (built-in function) which copies the components of a structure according to name.

The difference between the **_ERR*, **_RET* and **_BRK* macros is as follows (failed stands for different error conditions; for further information refer to the COPYOUT_ERR example given above):

```
1. *_ERR(....., errorno);□
if (failed)□
error = errorno;2. *_RET(....., errorno)□
if (failed)□
return (errorno);3. *_BRK(....., errorno)□
if (failed) {□
error = errorno;□
break;□
}
```

The **_ELEM_** macros can be used to *copyin* / *copyout* a structure component:

```
if(copyin((caddr_t)&user_buf->e_int, (caddr_t)&kernel_buf->e_int,□
sizeof(int)) == 1) □
error = EFAULT;
```

can be replaced by

```
COPYIN_ELEM_ERR(user_buf, kernel_buf, struct example, e_int,□
EFAULT);
```

If *copyin* / *copyout* procedures are required for n structures, the old statement:

```
if (copyin((caddr_t)user_buf, (caddr_t)kernel_buf, count *□
sizeof(struct example))□
== -1)□
error = EFAULT;
```

can be replaced by:

```
NCOPYIN_ERR(user_buf, kernel_buf, struct example, count, EFAULT);
```

A statement like:

```
if (copyin((caddr_t)user_long, (caddr_t)kernel_long, sizeof(long)) □
== -1) □
... if (copyout((caddr_t)kernel_long, (caddr_t)user_long, sizeof(long)) □
== -1)□
...
```

can be replaced by:

```
if (COPYIN_LP(user_long, kernel_long))□
```

```
...
if (COPYOUT_LP(kernel_long, user_long))
```

Please note the following guidelines for other uses of *copyin* or *copyout*:

1. Include **<sys/compat32.h>** in your code.
2. Establish which data structure is to be copied into or out of the kernel. Arrays cannot be handled in this way; you must convert these elements yourself.
3. Add **#pragma compat32** to your program. This is syntactically similar to a type definition (typedef), but automatically converts basic types to the 32-bit size and alignment.
4. Carry out a test to see whether 32-bit or 64-bit data structures are required. If 64-bit data structures are required, copy the data structure in the usual way. Otherwise, complete the steps listed below.
5. Declare a 32-bit variable which was created by the *compat32* macro.
6. Use *__copybyname* or one of the macros mentioned above to convert your 64-bit data structure to a 32-bit data structure (or vice versa).
7. Make sure when converting to a 32-bit data structure that no truncations occur.
8. Call *copyin* / *copyout* using the 32-bit version of the data structure.

The internal compiler function *__copybyname* converts a union as a C construct. However, it uses the first component of a union to determine the appropriate conversion and issue a warning. If the first field of the union is not the correct field to be copied, you must correct this explicitly after the *__copybyname* call.

Think of *__copybyname* as a short version of the most frequently occurring cases, and that there may still be situations in which the *copyin* / *copyout* macros are not suitable. There may even be situations in which the short version provided by *__copybyname* does not even help.

5.4 #ifdefs of the 64-bit kernel

__LP64 __

This define statement is activated automatically by the compiler system when kernel components are generated. It is used to differentiate kernel code which is new **and** is incompatible on 32-bit and 64-bit operating systems. It is only used for new code when the new code could not be compiled or executed for a compilation in 32-bit mode. In other words, it is not needed for the following statement:

```
#if defined(__LP64__)
int   x;
#else
long   x;
#endif
```

In a 32-bit kernel, long is the same size as int. Given this, we recommend that you use long x, provided it does not cause any other problems.

In include files, the difference between the types int and long is always an incompatible change (e.g. for X/OPEN and other standards). The *#ifdef* statement must always be used in include files.

5.5 Structure of the kernel return values

The rval union, in which the return values for the system calls are stored, has been changed. Given that this structure is returned to the user in registers, the size of each field must be the same as the size of the register; otherwise, the data will not be aligned correctly. Previously, the union in a 32-bit kernel looked like this:

```
union rval {
struct  {
long    r_v1;
long    r_v2;
} r_v;
off_t   r_off;
time_t  r_time;
```

```
};
```

This structure was converted to:

```
union rval {
struct  {
long    r_v1;
long    r_v2;
} r_v;
off_t   r_off;
long    r_time;
};
```

5.6 DDI/DKI incompatibility

The kernel interface for *ioctl* routines specifies that the argument *arg* of *prefixioctl* is an integer. However, many drivers expect user programs to specify a pointer as the argument. This does not cause problems in a 32-bit environment since integers and pointers are the same size. In the 64-bit environment, however, the sizes of integers and pointers are different. This means that the type of the argument must be changed to long. All character and stream drivers must change the type of their *arg* arguments in their prototypes and function definitions.

Old definition:

```
int prefixioctl(dev_t dev, int cmd, int arg, int mode, cred_t *cr, long *rv);
```

New definition:

```
int prefixioctl(dev_t dev, int cmd, long arg, int mode, cred_t *cr, long *rv);
```