SYSTEM V APPLICATION BINARY INTERFACE

SPARC® Processor Supplement

Third Edition

Contents

Figures and Tables

1 INTRODUCTION

SPARC Processor and the System V ABI

The *System V Application Binary Interface*, or ABI, defines a system interface for compiled application programs. Its purpose is to establish a standard binary interface for application programs on systems that implement the interfaces defined in the *System V Interface Definition, Issue 3*.

This document is a supplement to the generic *System V ABI*, and it contains information specific to System V implementations built on the SPARC™ processor architecture. Together, these two specifications, the generic *System V ABI* and the *SPARC System V ABI Supplement*, constitute a complete *System V Application Binary Interface* specification for systems that implement the SPARC processor architecture.

How to Use the SPARC ABI Supplement

This document is a supplement to the generic *System V ABI* and contains information referenced in the generic specification that may differ when System V is implemented on different processors. Therefore, the generic ABI is the prime reference document, and this supplement is provided to fill gaps in that specification.

As with the *System V ABI*, this specification references other publicly-available reference documents, especially the *The SPARC™ Architecture Manual, Version 8* (Copyright (c) 1992, SPARC International, Inc., ISBN 0-13-825001-4). All the information referenced by this supplement should be considered part of this specification, and just as binding as the requirements and data explicitly included here.

Evolution of the ABI Specification

The *System V Application Binary Interface* will evolve over time to address new technology and market requirements, and will be reissued at intervals of approximately three years. Each new edition of the specification is likely to contain extensions and additions that will increase the potential capabilities of applications that are written to conform to the ABI.

As with the *System V Interface Definition*, the ABI will implement **Level 1** and **Level 2** support for its constituent parts. **Level 1** support indicates that a portion of the specification will continue to be supported indefinitely, while **Level 2** support means that a portion of the specification may be withdrawn or altered after the next edition of the ABI is made available. That is, a portion of the specification moved to **Level 2** support in an edition of the ABI specification will remain in effect at least until the following edition of the specification is published.

These **Level 1** and **Level 2** classifications and qualifications apply to this Supplement, as well as to the generic specification. All components of the ABI and of this supplement have **Level 1** support unless they are explicitly labelled as **Level 2**.

2 SOFTWARE INSTALLATION

Software Distribution Formats

Physical Distribution Media

Approved media for physical distribution of ABI-conforming software are listed below. Inclusion of a particular medium on this list does not require an ABI-conforming system to accept that medium. For example, a conforming system may install all software through its network connection and accept none of the listed media.

- 3.5" floppy disk: double-sided, 80 cylinders / side, 18 sectors / cylinder, 512 bytes / sector.
- 150 MB quarter-inch cartridge tape in QIC-150 format.

The QIC-150 cartridge tape data format is described in *Serial Recorded Magnetic Tape Cartridge for Information Interchange, Eighteen Track 0.250 in. (6.30 mm) 10,000 bpi (394 bpmm) Streaming Mode Group Code Recording*, Revision 1, May 12, 1987. This document is available from the Quarter-Inch Committee (QIC) through Freeman Associates, 311 East Carillo St., Santa Barbara, CA 93101.

3 LOW-LEVEL SYSTEM INFORMATION

Machine Interface

Processor Architecture

The SPARC Architecture Manual (Version 8) defines the processor architecture. Programs intended to execute directly on the processor use the instruction set, instruction encodings, and instruction semantics of the architecture. Three points deserve explicit mention.

- A program may assume all documented instructions exist.
- A program may assume all documented instructions work.
- A program may use only the instructions defined by the architecture.

In other words, *from a program's perspective*, the execution environment provides a complete and working implementation of the SPARC architecture.

This does not imply that the underlying implementation provides all instructions in hardware, only that the instructions perform the specified operations and produce the specified results. The ABI neither places performance constraints on systems nor specifies what instructions must be implemented in hardware.

Some processors might support the SPARC architecture as a subset, providing additional instructions or capabilities. Programs that use those capabilities explicitly do not conform to the SPARC ABI. Executing those programs on machines without the additional capabilities gives undefined behavior.

Data Representation

Within this specification, the term *halfword* refers to a 16-bit object, the term *word* refers to a 32-bit object, and the term *doubleword* refers to a 64-bit object.

Fundamental Types

Figure 3-1 shows the correspondence between ANSI C's scalar types and the processor's.

Figure 3-1: Scalar Types

Type	C	sizeof	Alignment (bytes)	SPARC
Integral	`char` `signed char`	1	1	signed byte
	`unsigned char`	1	1	unsigned byte
	`short` `signed short`	2	2	signed halfword
	`unsigned short`	2	2	unsigned halfword

	`int`
	`signed int`
	`long`
	`signed long`
	`enum`
		4	4	signed word

	`unsigned int` `unsigned long`	4	4	unsigned word
Pointer	*any-type* `*` *any-type* `(*)()`	4	4	unsigned word
Floating-point	`float`	4	4	single-precision
	`double`	8	8	double-precision
	`long double`	16	8	quad-precision

A null pointer (for all types) has the value zero.

Double- and quad-precision values occupy 1 and 2 doublewords, respectively. Their natural alignment is a doubleword boundary, meaning their addresses are multiples of 8. Compilers should allocate independent data objects with the proper alignment; examples include global arrays of double-precision variables, FORTRAN `COMMON` blocks, and unconstrained stack objects. However, some language facilities (such as FORTRAN `EQUIVALENCE` statements) and the function calling sequence may create objects with only word alignment. Consequently, arbitrary double- and quad-precision addresses, such as pointers or reference parameters, might or might not be properly aligned. When a compiler knows an address is aligned properly, it can use load and store doubleword instructions; otherwise, it must load and store the object one word at a time.

Aggregates and Unions

Aggregates (structures and arrays) and unions assume the alignment of their most strictly aligned component. The size of any object, including aggregates and unions, always is a multiple of the object's alignment. An array uses the same alignment as its elements. Structure and union objects can require padding to meet size and alignment constraints. The contents of any padding is undefined.

- An entire structure or union object is aligned on the same boundary as its most strictly aligned member.
- Each member is assigned to the lowest available offset with the appropriate alignment. This may require *internal padding*, depending on the previous member.

- A structure's size is increased, if necessary, to make it a multiple of the alignment. This may require *tail padding*, depending on the last member.

In the following examples, members' byte offsets appear in the upper left corners.

Figure 3-2: Structure Smaller Than a Word

```
struct {
    char   c;
};
```

Byte aligned, `sizeof` is 1

0	c

Figure 3-3: No Padding

```
struct {
    char   c;
    char   d;
    short  s;
    long   n;
};
```

Word aligned, `sizeof` is 8

0	c	1	d	2	s
4	n				

Figure 3-4: Internal Padding

```
struct {
    char   c;
    short  s;
};
```

Halfword aligned, `sizeof` is 4

0	c	1	*pad*
2	s		

Figure 3-5: Internal and Tail Padding

```
struct {
    char   c;
    double d;
    short  s;
};
```

Doubleword aligned, `sizeof` is 24

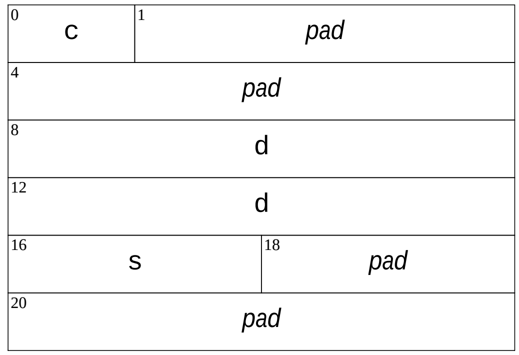
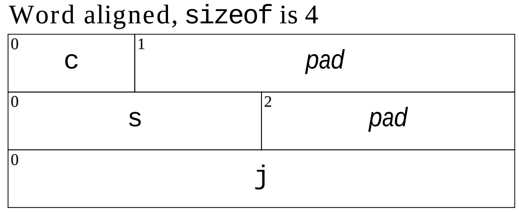


Figure 3-6: `union` Allocation

```
union {
    char   c;
    short  s;
    int    j;
};
```

Word aligned, `sizeof` is 4



Bit-Fields

C `struct` and `union` definitions may have *bit-fields*, defining integral objects with a specified number of bits.

Figure 3-7: Bit-Field Ranges

Bit-field Type	Width w	Range
`signed char`	1 to 8	-2^{w-1} to $2^{w-1}-1$
`char`		0 to $2^{w}-1$
`unsigned char`		0 to $2^{w}-1$
`signed short`	1 to 16	-2^{w-1} to $2^{w-1}-1$
`short`		0 to $2^{w}-1$
`unsigned short`		0 to $2^{w}-1$
`signed int`	1 to 32	-2^{w-1} to $2^{w-1}-1$
`int`		0 to $2^{w}-1$
`enum`		0 to $2^{w}-1$
`unsigned int`		0 to $2^{w}-1$

Figure 3-7: Bit-Field Ranges (continued)

`signed long`	1 to 32	-2^{w-1} to $2^{w-1}-1$
`long`		0 to $2^{w}-1$
`unsigned long`		0 to $2^{w}-1$

''Plain'' bit-fields always have non-negative values. Although they may have type `char`, `short`, `int`, or `long` (which can have negative values), these bit-fields are extracted into a word with zero fill. Bit-fields obey the same size and alignment rules as other structure and union members, with the following additions.

- Bit-fields are allocated from left to right (most to least significant).
- A bit-field must entirely reside in a storage unit appropriate for its declared type. Thus a bit-field never crosses its unit boundary.
- Bit-fields may share a storage unit with other `struct/union` members, including members that are not bit-fields. Of course, `struct` members occupy different parts of the storage unit.
- Unnamed bit-fields' types do not affect the alignment of a structure or union, although individual bit-fields' member offsets obey the alignment constraints.

The following examples show `struct` and `union` members' byte offsets in the upper left corners; bit numbers appear in the lower corners.

Figure 3-8: Bit Numbering

`0x01020304`

Byte offset	0	1	2	3
Value	01	02	03	04
Bits	31	23	15	7 … 0

Figure 3-9: Left-to-Right Allocation

```
struct {
    int    j:5;
    int    k:6;
    int    m:7;
};
```

Word aligned, `sizeof` is 4

Byte offset	0			
Member	j	k	m	*pad*
Bits	31	26	20	13 … 0

Figure 3-10: Boundary Alignment

```
struct {
    short  s:9;
    int    j:9;
    char   c;
    short  t:9;
    short  u:9;
    char   d;
};
```

Word aligned, `sizeof` is 12

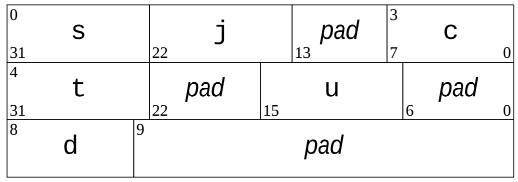


Figure 3-11: Storage Unit Sharing

```
struct {
    char   c;
    short  s:8;
};
```

Halfword aligned, `sizeof` is 2

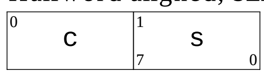


Figure 3-12: `union` Allocation

```
union {
    char   c;
    short  s:8;
};
```

Halfword aligned, `sizeof` is 2

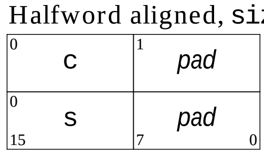
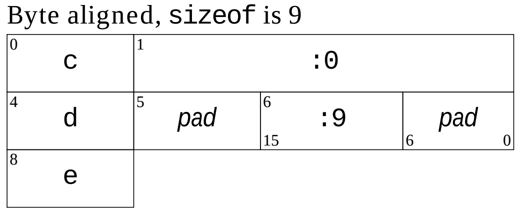


Figure 3-13: Unnamed Bit-Fields

```
struct {
    char   c;
    int    :0;
    char   d;
    short  :9;
    char   e;
    char   :0;
};
```

Byte aligned, `sizeof` is 9



As the examples show, `int` bit-fields (including `signed` and `unsigned`) pack more densely than smaller base types. One can use `char` and `short` bit-fields to force particular alignments, but `int` generally works better.

Function Calling Sequence

This section discusses the standard function calling sequence, including stack frame layout, register usage, parameter passing, and so on. The system libraries described in Chapter 6 require this calling sequence.

NOTE C programs follow the conventions given here. For specific information on the implementation of C, see ''Coding Examples'' in this chapter.

Registers and the Stack Frame

SPARC provides 32 floating-point registers and 8 integer registers that are global to a running program, as the `save` and `restore` instructions do not affect them. All remaining integer registers are windowed: 24 are visible at any time, and sets of 24 overlap by 8 registers each. The `save` and `restore` instructions manipulate the windows as part of the normal function prologue and epilogue, making the caller's 8 *out* registers coincide with the callee's 8 *in* registers. Each window set also has 8 unshared *local* registers. Generally, each new frame on the dynamic call stack uses a new register window.

Brief register descriptions appear in Figures 3-14 and 3-15; more complete information appears later.

Figure 3-14: A Function's Window Registers

Type	Name			Usage
in		%i7	%r31	return address – 8 †
	%fp,	%i6	%r30	frame pointer †
		%i5	%r29	incoming param 5 †
		%i4	%r28	incoming param 4 †
		%i3	%r27	incoming param 3 †
		%i2	%r26	incoming param 2 †
		%i1	%r25	incoming param 1 †
		%i0	%r24	incoming param 0, † outgoing return value
local		%l7	%r23	local 7 †
		%l6	%r22	local 6 †
		%l5	%r21	local 5 †
		%l4	%r20	local 4 †
		%l3	%r19	local 3 †
		%l2	%r18	local 2 †
		%l1	%r17	local 1 †
		%l0	%r16	local 0 †
out		%o7	%r15	address of `call` instruction, ‡ temporary value
	%sp,	%o6	%r14	stack pointer †
		%o5	%r13	outgoing param 5 ‡
		%o4	%r12	outgoing param 4 ‡
		%o3	%r11	outgoing param 3 ‡
		%o2	%r10	outgoing param 2 ‡
		%o1	%r9	outgoing param 1 ‡
		%o0	%r8	outgoing param 0, ‡ incoming return value

Figure 3-15: A Function's Global Registers

Type	Name		Usage
	%g7	%r7	global 7 (reserved for system)
	%g6	%r6	global 6 (reserved for system)
	%g5	%r5	global 5 (reserved for system)
	%g4	%r4	global 4 (reserved for application)
	%g3	%r3	global 3 (reserved for application)
	%g2	%r2	global 2 (reserved for application)
	%g1	%r1	global 1 ‡

Figure 3-15: A Function's Global Registers (continued)

global	%g0	%r0	0
floating-point		%f31	floating-point value ‡
		. . .	. . .
		%f0	floating-point value, ‡ floating-point return value
special		%y	Y register ‡

NOTE Registers marked † above are assumed to be preserved across a function call. Registers marked ‡ above are assumed to be destroyed (volatile) across a function call.

In addition to a register window, each function has a frame on the run-time stack. This stack grows downward from high addresses, moving in parallel with the current register window. Figure 3-16 shows the stack frame organization. Several key points about the stack frame deserve mention.

- Although the architecture requires only word alignment, software convention and the operating system require every stack frame to be doubleword aligned.
- Every stack frame must have a 16-word save area for the *in* and *local* registers, in case of window overflow or underflow. This save area always must exist at `%sp+0`.
- Software convention requires space for the `struct`/`union` return value pointer, even if the word is unused.
- Although the first 6 words of arguments reside in registers, the standard stack frame reserves space for them. ''Coding Examples'' below explains how these words may be used to implement variable argument lists. Arguments beyond the sixth reside on the stack.
- Other areas depend on the compiler and the code being compiled. The standard calling sequence does not define a maximum stack frame size, nor does it restrict how a language system uses the ''unspecified'' areas of the standard stack frame.

Figure 3-16: Standard Stack Frame

Base	Offset	Contents	Frame
		unspecified . . . variable size	*High addresses*
%fp	+92	(if present) incoming arguments 6, . . .	Previous
%fp	+68	six words into which function may write incoming arguments 0 to 5	
%fp	+64	`struct`/`union` return pointer	

Figure 3-16: Standard Stack Frame (continued)

`%fp`	0	16-word window save area	
`%fp`	–1	unspecified . . . variable size	Current
`%sp`	+92	(if needed) outgoing arguments 6, . . .	
`%sp`	+68	six words into which callee may write outgoing arguments 0 to 5	
`%sp`	+64	`struct`/`union` return pointer	
`%sp`	0	16-word window save area	*Low addresses*

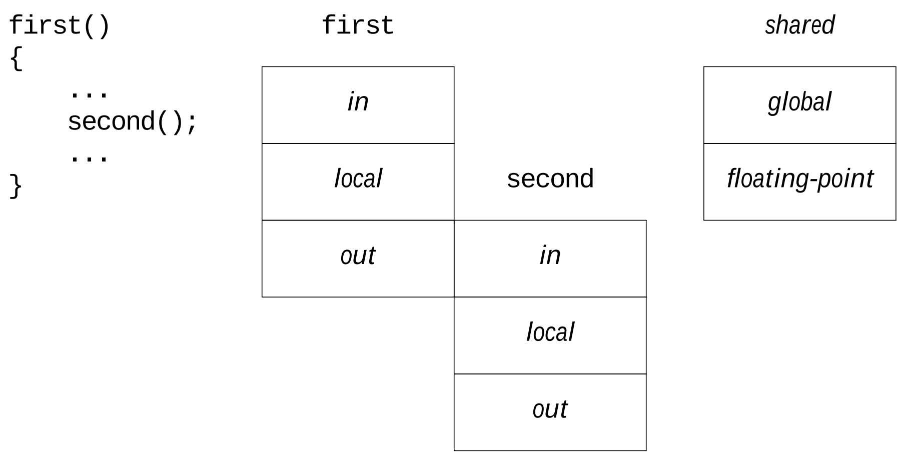
Across function boundaries, the standard function prologue shifts the register window, making the calling function's *out* registers the called function's *in* registers. It also allocates stack space, including the required areas of Figure 16 and any private space it needs. The lowest 16 words in the stack must—at all times—be reserved as the register save area. The example below illustrates this and allocates 80 bytes for the stack frame.

Figure 3-17: Function Prologue

```
second:
        save    %sp, -80, %sp
```

For demonstration, assume a function named `first` calls `second`. The register windows for the two functions appear below.

Figure 3-18: Register Windows



As explained later, the function epilogue executes a `restore` instruction to unwind the stack and restore the register windows to their original condition.

NOTE Strictly speaking, a function does not need the save and `restore` instructions if it preserves the registers as described below. Although some functions can be optimized to eliminate the save and `restore`, the general case uses the standard prologue and epilogue.

Some registers have assigned roles.

`%sp` or `%o6` — The *stack pointer* holds the limit of the current stack frame, which is the address of the stack's bottommost, valid word. Stack contents below the stack pointer are undefined. At all times the stack pointer must point to a doubleword aligned, 16-word window save area.

`%fp` or `%i6` — The *frame pointer* holds the address of the previous stack frame, which coincides with the word immediately above the current frame. Consequently, a function has registers pointing to both ends of its frame. Incoming arguments reside in the previous frame, referenced as positive offsets from `%fp`.

`%i0` and `%o0` — *Integral and pointer return values* appear in `%i0`. A function that returns a structure, union, or quad-precision value places the address of the result in `%i0`. A calling function receives values in the coincident *out* register, `%o0`.

`%i7` and `%o7` — The *return address* is the location to which a function should return control. Because a calling function's *out* registers coincide with the called function's *in* registers, the calling function puts a return address in its own `%o7`, while the called function finds the return address in `%i7`.

Actually, the return address register holds the call instruction's address, normally making the return address `%i7+8` for the called function. (Every call instruction has a delay instruction.) Between function calls, `%o7` serves as a scratch register.

`%f0` and `%f1`	*Floating-point return values* appear in the floating-point registers. Single-precision values occupy `%f0`; double-precision values occupy `%f0` and `%f1`. Otherwise, these are scratch registers.
`%i0` through `%i5`	*Incoming parameters* use up to 6 *in* registers. Arguments beyond the sixth word appear on the stack, as explained above. See the discussion below on structures, unions, and floating-point values.
`%o0` through `%o5`	*Outgoing arguments* use up to 6 *out* registers. Argument words beyond the sixth are written onto the stack.
`%l0` through `%l7`	*Local registers* have no specified role in the standard calling sequence.
`%f0` through `%f31`	Except for floating-point return values, *global floating-point registers* have no specified role in the standard calling sequence.
`%g0` and `%g1`	*Global integer registers* 0 and 1 have no specified role in the standard calling sequence.
`%g2` through `%g4`	*Global integer registers* 2, 3, and 4 are reserved for the application software. System software (including the libraries described in Chapter 6) preserves these registers' values for the application. Their use is intended to be controlled by the compilation system and must be consistent throughout the application.
`%g5` through `%g7`	*Global integer registers* 5, 6, and 7 are reserved for system software. Because system software provides the low-level operating system interface, including signal handling, an application cannot change the registers and safely preserve the system values, even by saving and restoring them across function calls. Therefore, application software must not change these registers' values.
`%y`	The *Y register* has no specified role in the standard calling sequence.

With some exceptions given below, all registers visible to both a calling and a called function ‘‘belong’’ to the called function. In other words, a called function may use all visible registers without saving their values before it changes them, and without restoring their values before it returns. Registers in this category include *global*, *floating-point*, *out* (for the calling function), *in* (for the called function), the Y register, the processor state register (PSR), and the floating-point state register (FSR). Correspondingly, if a calling function wants to preserve such a register value across a function call, it must save the value and restore it explicitly. *Local* registers in each window are private. A called function should not change its calling function's *local* or *in* registers, even though the registers may be visible temporarily. The exceptions are the stack pointer, `%sp`, and global registers `%g5` through `%g7`. A called function is obligated to preserve the stack pointer for its caller; application programs must never change the system global registers.

Signals can interrupt processes [see `signal`(BA_OS)]. Functions called during signal handling have no unusual restrictions on their use of registers. Moreover, if a signal handling function returns, the process resumes its original execution path with registers restored to their original values. Thus programs and compilers may freely use all registers, even *global* and *floating-point* registers, without the danger of signal handlers changing their values.

Integral and Pointer Arguments

As mentioned, a function receives its first 6 argument words through the *in* registers: `%i0` is the first, `%i1` is the second, and so on. Functions pass all integer-valued arguments as words, expanding signed or unsigned bytes and halfwords as needed. If a function call has more than 6 integral or pointer arguments, the others go on the stack.

Figure 3-19: Integral and Pointer Arguments

Call	Argument	Caller	Callee
g(1, 2, 3, 4, 5, 6, 7, (void *)0);	1	%o0	%i0
	2	%o1	%i1
	3	%o2	%i2
	4	%o3	%i3
	5	%o4	%i4
	6	%o5	%i5
	7	%sp+92	%fp+92
	(void *)0	%sp+96	%fp+96

Floating-Point Arguments

The integer *in* registers also hold floating-point arguments: single-precision values use one register and double-precision use two. See the following section for information on quad-precision values. As floating-point operations cannot use the integer registers, compilers normally store the input registers to the stack before operating on floating-point values. See ''Coding Examples'' for information about floating-point arguments and variable argument lists. The example below uses only double-precision arguments. Single-precision arguments behave similarly.

Figure 3-20: Floating-Point Arguments

Call	Argument	Caller	Callee
h(1.414, 1, 2.998e10, 2.718);	word 0, 1.414	%o0	%i0
	word 1, 1.414	%o1	%i1
	1	%o2	%i2
	word 0, 2.998e10	%o3	%i3
	word 1, 2.998e10	%o4	%i4
	word 0, 2.718	%o5	%i5
	word 1, 2.718	%sp+92	%fp+92

Structure, Union, and Quad-Precision Arguments

As described in the data representation section, structures and unions can have byte, halfword, word, or doubleword alignment, depending on the constituents. To ensure proper argument alignment and to facilitate addressing, structure and union objects are not passed directly in the argument list. Quad-precision values follow the same conventions as structures and unions.

The example below shows the *effect* only; C code does *not* change.

Figure 3-21: Sending Structure, Union, and Quad-Precision Arguments

Source

```
caller()
{
        struct s s;

        callee(s);
}
```

Compiler's Internal Form

```
caller()
{
        struct s s, s2;

        s2 = s;
        callee(&s2);
}
```

Addresses occupy one word; so structure, unions, and quad-precision values occupy a single word as function arguments. In this respect, these arguments behave the same as integral and pointer arguments, described above. The example's temporary copy of the object, `s2` above, provides call-by-value semantics, letting the called function modify its arguments without affecting the calling function's object, `s` above.

Because the calling function passes a pointer in the argument list, the compiled code for the called function must accept the same. Underlying machinations are transparent to the source program. The compiler translates appropriately, implicitly dereferencing the pointer as needed. Code for a called function might appear as follows. Again, the example below shows the *effect* only; C code does *not* change.

Figure 3-22: Receiving Structure, Union, and Quad-Precision Arguments

Source

```
callee(struct s arg)
{
        struct s s, s2;

        s.m = arg.m;
        s2 = arg;
}
```

Compiler's Internal Form

```
callee(struct s *arg)
{
        struct s s, s2;

        s.m = arg->m;
        s2 = *arg;
}
```

Functions Returning Scalars or No Value

A function that returns an integral or pointer value places its result in `%i0`; the calling function finds that value in `%o0`.

A floating-point return value appears in the floating-point registers for both the calling and the called function. Single-precision uses `%f0`; double-precision uses `%f0` and `%f1`; quad-precision uses the same method as structures and unions, described below.

Functions that return no value (also called procedures or `void` functions) put no particular value in any return register. Those registers may be used as scratch registers, however.

A call instruction writes its own address into *out* register `%o7`. As usual for a control transfer instruction, the call instruction takes a delay instruction that is executed before the first instruction of the called function. Because every instruction is one word long, the return address is the address of the call instruction plus 8. This value is `%i7+8` for the called function and `%o7+8` for the calling function. The following example returns the value contained in *local* register `%l4`.

Figure 3-23: Function Epilogue

```
jmpl     %i7 + 8, %g0
restore  %l4, 0, %o0
```

If a function returns no value, or if the return register already contains the desired value, the next epilogue would suffice.

Figure 3-24: Alternative Function Epilogue

```
jmpl     %i7 + 8, %g0
restore  %g0, 0, %g0
```

Functions Returning Structures, Unions, or Quad-Precision Values

As shown above, every stack frame reserves the word at `%fp+64`. If a function returns a structure, union, or quad-precision value, this word should hold the address of the object into which the return value should be copied. The caller provides space for the return value and places its address in the stack frame (the word is at `%sp+64` for the caller). Having the caller supply the return object's space allows re-entrancy.

NOTE Structures and unions in this context have fixed sizes. The ABI does not specify how to handle variable sized objects.

A function returning a structure, union, or quad-precision value also sets `%i0` to the value it finds in `%fp+64`. Thus when the caller receives control again, the address of the returned object resides in register `%o0`.

Both the calling and the called functions must cooperate to pass the return value successfully:

- The calling function must supply space for the return value and pass its address in the stack frame;
- The called function must use the address from the frame and copy the return value to the object so supplied.

Failure of either side to meet its obligations leads to undefined program behavior. The standard function calling sequence includes a method to detect such failures and to detect type mismatches.

Whenever a calling function expects a structure, union, or quad-precision return value from the function being called, the compiler generates an `unimp` (unimplemented) instruction immediately following the delay instruction of the call. The `unimp` instruction's immediate field holds the low-order 12 bits of the expected return value's size (higher bits are masked if the object is larger than 4095 bytes). When preparing to return its value, the called function checks for the presence of the `unimp` instruction, and it checks that the low-order 12 bits agree with the low-order 12 bits of the size it plans to copy. If all tests pass, the function copies the value and returns to `%i7+12`, skipping the call instruction, the delay instruction, and the `unimp` instruction.

If the called function disagrees with the caller's object size, it returns to `%i7+8`, executes the `unimp` instruction and causes an illegal instruction trap. If the called function does not return a structure, union, or quad-precision value, it will return to `%i7+8`, trapping similarly. See section ''Trap Interface'' in this chapter for more information about traps.

Finally, if the called function returns a structure, union, or quad-precision value but the calling function doesn't expect one, the called function copies nothing, returns to `%i7+8`, and continues executing (there will be no `unimp` instruction). Of course, the caller should assume no return value is present; both `%i0` and `%f0` have unpredictable values in this case. The following example assumes the return object has already been copied and its address is in *local* register `%l4`.

Figure 3-25: Function Epilogue

```
jmpl     %i7 + 12, %g0
restore  %l4, 0, %o0
```

Operating System Interface

Virtual Address Space

Processes execute in a 32-bit virtual address space. Memory management hardware translates virtual addresses to physical addresses, hiding physical addressing and letting a process run anywhere in the system's real memory. Processes typically begin with three logical segments, commonly called text, data, and stack. As Chapter 5 describes, dynamic linking creates more segments during execution, and a process can create additional segments for itself with system services.

Page Size

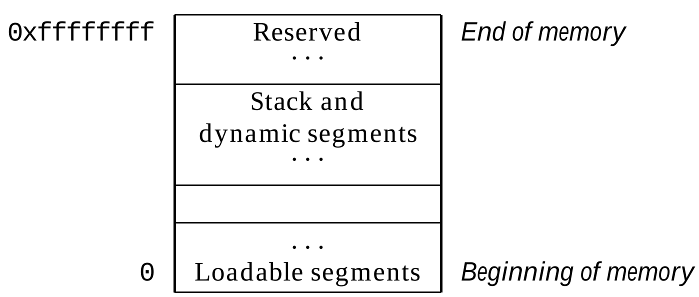
Memory is organized by pages, which are the system's smallest units of memory allocation. Page size can vary from one system to another, depending on the processor, memory management unit and system configuration. Processes may call `sysconf`(BA_OS) to determine the system's current page size. The maximum page size for SPARC is 64 KB.

Virtual Address Assignments

Conceptually, processes have the full 32-bit address space available. In practice, however, several factors limit the size of a process.

- The system reserves a configuration-dependent amount of virtual space.
- A tunable configuration parameter limits process size.
- A process whose size exceeds the system's available, combined physical memory and secondary storage cannot run. Although some physical memory must be present to run any process, the system can execute processes that are bigger than physical memory, paging them to and from secondary storage. Nonetheless, both physical memory and secondary storage are shared resources. System load, which can vary from one program execution to the next, affects the available amounts.

Figure 3-26: Virtual Address Configuration



Loadable segments
: Processes' loadable segments may begin at `0`. The exact addresses depend on the executable file format [see Chapters 4 and 5].

Stack and dynamic segments
: A process's stack and dynamic segments reside below the reserved area. Processes can control the amount of virtual memory allotted for stack space, as described below.

Reserved A reserved area resides at the top of virtual space.

NOTE Although application programs may begin at virtual address 0, they conventionally begin above `0x10000` (64 K), leaving the initial 64 K with an invalid address mapping. Processes that reference this invalid memory (for example, by dereferencing a null pointer) generate an access exception trap, as described in the ''Trap Interface'' section of this chapter. A process may, however, establish a valid mapping for this area using the `mmap`(KE_OS) facilities.

As the figure shows, the system reserves the high end of virtual space, with a process's stack and dynamic segments below that. Although the exact boundary between the reserved area and a process depends on the system's configuration, the reserved area shall not consume more than 512 MB from the virtual address space. Thus the user virtual address range has a minimum upper bound of `0xdfffffff`. Individual systems may reserve less space, increasing processes' virtual memory range. More information follows in the section ''Managing the Process Stack.''

Although applications may control their memory assignments, the typical arrangement follows the diagram above. Loadable segments reside at low addresses; dynamic segments occupy the higher range. When applications let the system choose addresses for dynamic segments (including shared object segments), it chooses high addresses. This leaves the ''middle'' of the address spectrum available for dynamic memory allocation with facilities such as `malloc`(BA_OS).

Managing the Process Stack

Section ''Process Initialization'' in this chapter describes the initial stack contents. Stack addresses can change from one system to the next—even from one process execution to the next on the same system. Processes, therefore, should *not* depend on finding their stack at a particular virtual address.

A tunable configuration parameter controls the system maximum stack size. A process also can use `setrlimit`(BA_OS), to set its own maximum stack size, up to the system limit. On SPARC, the stack segment has read, write, and execute permissions.

Coding Guidelines

Operating system facilities, such as `mmap`(KE_OS), allow a process to establish address mappings in two ways. First, the program can let the system choose an address. Second, the program can force the system to use an address the program supplies. This second alternative can cause application portability problems, because the requested address might not always be available. Differences in virtual address space can be particularly troublesome between different architectures, but the same problems can arise within a single architecture.

Processes' address spaces typically have three segment areas that can change size from one execution to the next: the stack [through `setrlimit`(BA_OS)], the data segment [through `malloc`(BA_OS)], and the dynamic segment area [through `mmap`(KE_OS)]. Changes in one area may affect the virtual addresses available for another. Consequently, an address that is available in one process execution might not be available in the next. A program that used `mmap`(KE_OS) to request a mapping at a specific address thus could appear to work in some environments and fail in others. For this reason, programs that wish to establish a mapping in their address space should let the system choose the address.

Despite these warnings about requesting specific addresses, the facility can be used properly. For example, a multiprocess application might map several files into the address space of each process and build relative pointers among the files' data. This could be done by having each process ask for a certain amount of memory at an address chosen by the system. After each process receives its own, private address from the system, it would map the desired files into memory, at specific addresses within the original area. This collection of mappings could be at different addresses in each process but their *relative* positions would be fixed. Without the ability to ask for specific addresses, the application could not build shared data structures, because the relative positions for files in each process would be unpredictable.

Trap Interface

Two execution modes exist in the SPARC architecture: user and supervisor. Processes run in user mode, and the operating system kernel runs in supervisor mode. As the SPARC architecture manual describes, the processor changes mode to handle *traps*, which may be precise, interrupting or deferred. Precise and deferred traps, being caused by instruction execution, can be explicitly generated by a process. This section, therefore, specifies those trap types with defined behavior.

Hardware Trap Types

The operating system defines the following correspondence between hardware traps and the signals specified by `signal`(BA_OS).

Figure 3-27: Hardware Traps and Signals

Trap Name	Signal
cp_disabled	SIGILL
cp_exception	SIGILL
data_access_error	unspecified
data_access_exception	SIGSEGV, SIGBUS
data_store_error	unspecified
division_by_zero	SIGFPE
fp_disabled	SIGILL
fp_exception	SIGFPE
illegal_instruction	SIGILL
instruction_access_exception	SIGSEGV, SIGBUS
mem_address_not_aligned	SIGBUS
privileged_instruction	SIGILL
r_register_access_error	unspecified
tag_overflow	SIGEMT
trap_instruction	see next table
window_overflow	none
window_underflow	none

Two trap types, instruction_access_exception and data_access_exception, can generate two signals. In both cases, the ‘‘normal’’ signal is SIGSEGV. Nonetheless, if the access also causes some external memory error (such as a parity error), the system generates SIGBUS.

Floating-point instructions exist in the architecture, but they may be implemented either in hardware or software. If the fp_disabled or fp_exception trap occurs because of an unimplemented, valid instruction, the process receives no signal. Instead, the system intercepts the trap, emulates the instruction, and returns control to the process. A process receives SIGILL for the fp_disabled trap only when the indicated floating-point instruction is illegal (invalid encoding, and so on).

Software Trap Types

The operating system defines the following correspondence between software traps and the signals specified by `signal`(BA_OS).

Figure 3-28: Software Trap Types

Trap Number	Signal	Purpose
0	`SIGSYS`	System calls
1	`SIGTRAP`	Breakpoints
2	`SIGFPE`	Division by zero
3	none	Flush windows
4	none	Clean windows
5	`SIGILL`	Range checking
6	none	Fix alignment
7	`SIGFPE`	Integer overflow
8	`SIGSYS`	System calls
9-15	unspecified	Reserved for the operating system
16-31	`SIGILL`	Unspecified
32	none	Get integer condition codes
33	none	Set integer condition codes
34-127	unspecified	Reserved for the operating system

0 and 8 System calls, or requests for operating system services, use a type 0 or 8 trap instruction for the low-level implementation. Normally, system calls do not generate a signal, but `SIGSYS` can occur in some error conditions. Both trap numbers are reserved, and they are not (necessarily) equivalent.

NOTE The ABI does not define the implementation of individual system calls. Instead, programs should use the system libraries that Chapter 6 describes. Programs with embedded system call trap instructions do not conform to the ABI.

1 A debugger can set a breakpoint by inserting a trap instruction whose type is 1.

2 A process can explicitly signal division by zero with this trap.

3 By executing a type 3 trap, a process asks the system to flush all its register windows to the stack.

4 Normally during process execution, save instructions allocate new register windows with undefined *local* and *out* register contents. Executing a type 4 trap causes the system to initialize *local* and *out* registers in all subsequent new windows either to zero or to a valid program counter value. In addition, new windows allocated when a save instruction generates a window_overflow trap are also initialized in this manner. This behavior continues until the process terminates.

5 A process can explicitly signal a range checking error with this trap.

6 Executing a type 6 trap makes the operating system ‘‘fix’’ subsequent unaligned data references. Although the references still generate memory_address_not_aligned traps, the operating system handles the trap, emulates the data references, and returns control to the process without generating a signal. In this context, a ‘‘data reference’’ is a load or a store operation. Implicit memory references, such as control transfers, must always be aligned properly, and the stack must always be aligned as described elsewhere.

7 A process can explicitly signal integer overflow with this trap. Either a positive or a negative value can cause overflow.

9 to 15 The operating system reserves these trap types for its own use. Programs that use them do not conform to the ABI.

16 to 31 Software trap types in this range have no specified meaning; moreover, they will never be specified. Thus these trap types are reserved for process-specific, machine-specific, and system-specific purposes. Besides receiving signal `SIGILL` for these traps, the signal handler receives the trap type (16-31) as the signal code.

32 Executing a type 32 trap instruction copies the integer condition codes from the PSR to *global* register `%g1`. The result is right-justified; other `%g1` bits are set to zero.

33 Executing a type 33 trap instruction copies the rightmost four bits from *global* register `%g1` to the PSR integer condition codes. Other bits in `%g1` are ignored.

34 to 127 The operating system reserves these trap types for its own use. Programs that use them do not conform to the ABI.

Process Initialization

This section describes the machine state that `exec`(BA_OS) creates for ‘‘infant’’ processes, including argument passing, register usage, stack frame layout, and so on. Programming language systems use this initial program state to establish a standard environment for their application programs. As an example, a C program begins executing at a function named `main`, conventionally declared in the following way.

Figure 3-29: Declaration for `main`

```
extern int main(int argc, char *argv[], char *envp[]);
```

Briefly, `argc` is a non-negative argument count; `argv` is an array of argument strings, with `argv[argc]==0`; and `envp` is an array of environment strings, also terminated by a null pointer.

Although this section does not describe C program initialization, it gives the information necessary to implement the call to `main` or to the entry point for a program in any other language.

Special Registers

As the architecture defines, two state registers control and monitor the processor: the processor state register (PSR) and the floating-point state register (FSR). Application programs cannot access the PSR directly; they run in the processor's *user mode*, and the instructions to read and write the PSR are privileged. Nonetheless, a program ''sees'' a processor that behaves as if the PSR had the following values. PSR fields not in the table either have unspecified values or do not affect user program behavior.

Figure 3-30: Processor State Register Fields

Field	Value	Note
icc	unspecified	Integer condition codes unspecified
EC	unspecified	Coprocessor not specified
S	0	Processes run in user mode
ET	1	Traps enabled

No standard coprocessor is specified by the ABI. Applications that directly execute coprocessor operate instructions do not conform to the ABI. Individual system implementations may use a coprocessor (to improve performance, for example), but such use of the coprocessor should be under the control of *system* software, not the application.

Similarly, ancillary state registers (ASR's) besides the Y register either are privileged or unspecified by the architecture. Applications thus may not execute the `rdasr` and `wrasr` instructions, with the exceptions of `rdy` and `wry`.

The architecture defines floating-point instructions, and those instructions work whether the processor has a hardware floating-point unit or not. (A system may provide hardware or software floating-point facilities.) Consequently, the EF bit in the PSR is unspecified, letting the system set it according to the hardware configuration. In either case, however, the processor presents a working floating-point implementation, including an FSR with the following initial values.

Figure 3-31: Floating-Point State Register Fields

Field	Value	Note
RD	0	Round to nearest
TEM	0	Floating-point traps not enabled
NS	0	Nonstandard mode off
ftt	unspecified	Floating-point trap type unspecified
qne	0	Floating-point queue is empty
fcc	unspecified	Floating-point condition codes unspecified
aexc	0	No accrued exceptions
cexc	0	No current exceptions

Process Stack and Registers

When a process receives control, its stack holds the arguments and environment from `exec`(BA_OS).

Figure 3-32: Initial Process Stack

	Unspecified	*High addresses*
	Information block, including argument strings environment strings auxiliary information . . . (size varies)	
	Unspecified	
	Null auxiliary vector entry	
	Auxiliary vector . . . (2-word entries)	
	0 word	
	Environment pointers . . . (one word each)	
	0 word	
	Argument pointers . . . (*Argument count* words)	
`%sp+64`	Argument count	
`%sp+0`	Window save area (16 words)	*Low addresses*

Argument strings, environment strings, and the auxiliary information appear in no specific order within the information block; the system makes no guarantees about their arrangement. The system also may leave an unspecified amount of memory between the null auxiliary vector entry and the beginning of the information block.

Except as shown below, global, floating-point, and window registers have unspecified values at process entry. Consequently, a program that requires registers to have specific values must set them explicitly during process initialization. It should *not* rely on the system to set all registers to zero.

`%g1` — A non-zero value gives a function pointer that the application should register with `atexit`(BA_OS). If %g1 contains zero, no action is required.

`%g2`, `%g3`, and `%g4` — These registers are currently set to zero. Future versions of the system might use the registers to hold special values; so applications should not depend on these registers' values.

`%fp` — The system marks the deepest stack frame by setting the frame pointer to zero. No other frame's `%fp` has a zero value.

`%sp` Performing its usual job, the stack pointer holds the address of the bottom of the stack, which is guaranteed to be doubleword aligned.

Every process has a stack, but the system defines *no* fixed stack address. Furthermore, a program's stack address can change from one system to another—even from one process invocation to another. Thus the process initialization code must use the stack address in `%sp`. Data in the stack segment at addresses below the stack pointer contain undefined values.

Whereas the argument and environment vectors transmit information from one application program to another, the auxiliary vector conveys information from the operating system to the program. This vector is an array of the following structures, interpreted according to the `a_type` member.

Figure 3-33: Auxiliary Vector

```
typedef struct
{
        int     a_type;
        union {
                long    a_val;
                void    *a_ptr;
                void    (*a_fcn)();
        } a_un;
} auxv_t;
```

Figure 3-34: Auxiliary Vector Types, `a_type`

Name	Value	a_un
AT_NULL	0	ignored
AT_IGNORE	1	ignored
AT_EXECFD	2	a_val
AT_PHDR	3	a_ptr
AT_PHENT	4	a_val
AT_PHNUM	5	a_val
AT_PAGESZ	6	a_val
AT_BASE	7	a_ptr
AT_FLAGS	8	a_val
AT_ENTRY	9	a_ptr

`AT_NULL` The auxiliary vector has no fixed length; instead its last entry's `a_type` member has this value.

AT_IGNORE — This type indicates the entry has no meaning. The corresponding value of a_un is undefined.

AT_EXECFD — As Chapter 5 describes, exec(BA_OS) may pass control to an interpreter program. When this happens, the system places either an entry of type AT_EXECFD or one of type AT_PHDR in the auxiliary vector. The entry for type AT_EXECFD uses the a_val member to contain a file descriptor open to read the application program's object file.

AT_PHDR — Under some conditions, the system creates the memory image of the application program before passing control to the interpreter program. When this happens, the a_ptr member of the AT_PHDR entry tells the interpreter where to find the program header table in the memory image. If the AT_PHDR entry is present, entries of types AT_PHENT, AT_PHNUM, and AT_ENTRY must also be present. See Chapter 5 in both the System V ABI and the processor supplement for more information about the program header table.

AT_PHENT — The a_val member of this entry holds the size, in bytes, of one entry in the program header table to which the AT_PHDR entry points.

AT_PHNUM — The a_val member of this entry holds the number of entries in the program header table to which the AT_PHDR entry points.

AT_PAGESZ — If present, this entry's a_val member gives the system page size, in bytes. The same information also is available through sysconf(BA_OS).

AT_BASE — The a_ptr member of this entry holds the base address at which the interpreter program was loaded into memory. See ‘‘Program Header’’ in the System V ABI for more information about the base address.

AT_FLAGS — If present, the a_val member of this entry holds one-bit flags. Bits with undefined semantics are set to zero.

AT_ENTRY — The a_ptr member of this entry holds the entry point of the application program to which the interpreter program should transfer control.

Other auxiliary vector types are reserved. Currently, no flag definitions exist for AT_FLAGS. Nonetheless, bits under the 0xff000000 mask are reserved for system semantics.

In the following example, the stack resides below 0xf8000000, growing toward lower addresses. The process receives three arguments.

- cp
- src
- dst

It also inherits two environment strings (this example is not intended to show a fully configured execution environment).

- HOME=/home/dir
- PATH=/home/dir/bin:/usr/bin:

Its auxiliary vector holds one non-null entry, a file descriptor for the executable file.

- 13

The initialization sequence preserves the stack pointer's doubleword alignment.

Figure 3-35: Example Process Stack

<table>
<tr><td></td><td>n</td><td>:</td><td>\0</td><td>pad</td></tr>
<tr><td></td><td>r</td><td>/</td><td>b</td><td>i</td></tr>
<tr><td></td><td>:</td><td>/</td><td>u</td><td>s</td></tr>
<tr><td>0xf7fffff0</td><td>/</td><td>b</td><td>i</td><td>n</td></tr>
<tr><td></td><td>/</td><td>d</td><td>i</td><td>r</td></tr>
<tr><td></td><td>h</td><td>o</td><td>m</td><td>e</td></tr>
<tr><td></td><td>T</td><td>H</td><td>=</td><td>/</td></tr>
<tr><td>0xf7ffffe0</td><td>r</td><td>\0</td><td>P</td><td>A</td></tr>
<tr><td></td><td>e</td><td>/</td><td>d</td><td>i</td></tr>
<tr><td></td><td>/</td><td>h</td><td>o</td><td>m</td></tr>
<tr><td></td><td>O</td><td>M</td><td>E</td><td>=</td></tr>
<tr><td>0xf7ffffd0</td><td>s</td><td>t</td><td>\0</td><td>H</td></tr>
<tr><td></td><td>r</td><td>c</td><td>\0</td><td>d</td></tr>
<tr><td></td><td>c</td><td>p</td><td>\0</td><td>s</td></tr>
<tr><td></td><td colspan="4">0</td></tr>
<tr><td>0xf7ffffc0</td><td colspan="4">0</td></tr>
<tr><td></td><td colspan="4">13</td></tr>
<tr><td></td><td colspan="4">2</td></tr>
<tr><td></td><td colspan="4">0</td></tr>
<tr><td>0xf7ffffb0</td><td colspan="4">0xf7ffffe2</td></tr>
<tr><td></td><td colspan="4">0xf7ffffd3</td></tr>
<tr><td></td><td colspan="4">0</td></tr>
<tr><td></td><td colspan="4">0xf7ffffcf</td></tr>
<tr><td>0xf7ffffa0</td><td colspan="4">0xf7ffffcb</td></tr>
<tr><td></td><td colspan="4">0xf7ffffc8</td></tr>
<tr><td>0xf7ffff98</td><td colspan="4">3</td></tr>
<tr><td>%sp+0, 0xf7ffff58</td><td colspan="4">Window save area
(16 words)</td></tr>
</table>

Coding Examples

This section discusses example code sequences for fundamental operations such as calling functions, accessing static objects, and transferring control from one part of a program to another. Previous sections discuss how a program may use the machine or the operating system, and they specify what a program may and may not assume about the execution environment. Unlike previous material, the information here illustrates how operations *may* be done, not how they *must* be done.

As before, examples use the ANSI C language. Other programming languages may use the same conventions displayed below, but failure to do so does *not* prevent a program from conforming to the ABI. Two main object code models are available.

- *Absolute code*. Instructions can hold absolute addresses under this model. To execute properly, the program must be loaded at a specific virtual address, making the program's absolute addresses coincide with the process's virtual addresses.
- *Position-independent code*. Instructions under this model hold relative addresses, *not* absolute addresses. Consequently, the code is not tied to a specific load address, allowing it to execute properly at various positions in virtual memory.

Size and performance considerations further require large and small position-independent models, giving three models total. Following sections describe the differences between these models. Code sequences for the three models (when different) appear together, allowing easier comparison.

NOTE Examples below show code fragments with various simplifications. They are intended to explain addressing modes, not to show optimal code sequences nor to reproduce compiler output.

NOTE When other sections of this document show assembly language code sequences, they typically show only the absolute versions. Information in this section explains how position-independent code would alter the examples.

Code Model Overview

When the system creates a process image, the executable file portion of the process has fixed addresses, and the system chooses shared object library virtual addresses to avoid conflicts with other segments in the process. To maximize text sharing, shared objects conventionally use position-independent code, in which instructions contain no absolute addresses. Shared object text segments can be loaded at various virtual addresses without having to change the segment images. Thus multiple processes can share a single shared object text segment, even though the segment resides at a different virtual address in each process.

Position-independent code relies on two techniques.

- Control transfer instructions hold addresses relative to the program counter (PC). A PC-relative branch or function call computes its destination address in terms of the current program counter, *not* relative to any absolute address.
- When the program requires an absolute address, it computes the desired value. Instead of embedding absolute addresses in the instructions, the compiler generates code to calculate an absolute address during execution.

Because the processor architecture provides PC-relative call and branch instructions, compilers can satisfy the first condition easily.

A *global offset table* provides information for address calculation. Position-independent object files (executable and shared object files) have a table in their data segment that holds addresses. When the system creates the memory image for an object file, the table entries are relocated to reflect the absolute virtual addresses as assigned for an individual process. Because data segments are private for each process, the table entries can change—unlike text segments, which multiple processes share.

Two position-independent models give programs a choice between more efficient code with some size restrictions and less efficient code without those restrictions. Because of the processor's architecture, a global offset table with no more than 2048 entries (8192 bytes) is more efficient than a larger one. Programs that need more entries must use the larger, more general code.

Position-Independent Function Prologue

This section describes the function prologue for position-independent code. A function's prologue first allocates the local stack space. Position-independent functions also set *local* register `%l7` to the global offset table's address, accessed with the symbol _GLOBAL_OFFSET_TABLE_. Because `%l7` is private for each function and preserved across function calls, a function calculates its value once at the entry.

NOTE As a reminder, this entire section contains examples. Using `%l7` is a convention, not a requirement; moreover, this convention is private to a function. Not only could other registers serve the same purpose, but different functions in a program could use different registers.

To explain the following, code before label `1:` updates the stack pointer as usual. The `call` instruction puts its own *absolute* address into register `%o7`. The next two instructions calculate the *offset* between the `call` instruction and the global offset table. Adding the `call` instruction's address to the computed offset gives the global offset table's absolute address.

NOTE When an instruction uses _GLOBAL_OFFSET_TABLE_, it sees the offset between the current instruction and the global offset table as the symbol value.

Figure 3-36: Position-Independent Function Prologue

```
name:
     save   %sp, -80, %sp
1:   call   2f
     sethi  %hi(_GLOBAL_OFFSET_TABLE_+(.-1b)), %l7
2:   or     %l7, %lo(_GLOBAL_OFFSET_TABLE_+(.-1b)), %l7
     add    %l7, %o7, %l7
```

Both large and small position-independent models use this prologue. All models use the same function epilogue.

Data Objects

This discussion excludes stack-resident objects, because programs always compute their virtual addresses relative to the stack and frame pointers. Instead, this section describes objects with static storage duration.

In the SPARC architecture, only load and store instructions access memory. Because instructions cannot hold 32-bit addresses directly, a program normally computes an address into a register. Symbolic references in absolute code put the symbols' values—or absolute virtual addresses—into instructions.

Figure 3-37: Absolute Load and Store

C	Assembly

`extern int src;`
`extern int dst;`
`extern int *ptr;`
`ptr = &dst;`
	`.global   src, dst, ptr`
	
	`sethi    %hi(dst), %o0`
	`or       %lo(dst), %o0`
	`sethi    %hi(ptr), %o1`
	`st       %o0, [%o1 + %lo(ptr)]`

`*ptr = src;`	`sethi    %hi(src), %o0`
	`ld       [%o0 + %lo(src)], %o0`
	`sethi    %hi(ptr), %o1`
	`ld       [%o1 + %lo(ptr)], %o1`
	`st       %o0, [%o1]`

Position-independent instructions cannot contain absolute addresses. Instead, instructions that reference symbols hold the symbols' offsets into the global offset table. Combining the offset with the global offset table address in `%l7` gives the absolute address of the table entry holding the desired address. A program whose global offset table has no more than 8192 bytes can use the small model, with a base address in `%l7` plus a 13-bit, signed offset.

NOTE When assembling position-independent code, a symbol's ''value'' is the offset into the global offset table, not its virtual address.

Figure 3-38: Small Model Position-Independent Load and Store

C	Assembly

`extern int src;`
`extern int dst;`
`extern int *ptr;`
`ptr = &dst;`
	`.global   src, dst, ptr`
	
	`ld       [%l7 + dst], %o0`
	`ld       [%l7 + ptr], %o1`
	`st       %o0, [%o1]`

Figure 3-38: Small Model Position-Independent Load and Store (continued)

C	Assembly

`*ptr = src;`	`ld [%l7 + src], %o0`
	`ld [%o0], %o0`
	`ld [%l7 + ptr], %o1`
	`ld [%o1], %o1`
	`st %o0, [%o1]`

The large model assumes no limit on global offset table size; it computes the table offset into one register and combines that with `%l7` to address the desired entry.

Figure 3-39: Large Model Position-Independent Load and Store

C	Assembly

`extern int src;`
`extern int dst;`
`extern int *ptr;`
	`.global src, dst, ptr`

`ptr = &dst;`	`sethi %hi(dst), %o0`
	`or %lo(dst), %o0`
	`ld [%l7 + %o0], %o0`
	`sethi %hi(ptr), %o1`
	`or %lo(ptr), %o1`
	`ld [%l7 + %o1], %o1`
	`st %o0, [%o1]`

`*ptr = src;`	`sethi %hi(src), %o0`
	`or %lo(src), %o0`
	`ld [%l7 + %o0], %o0`
	`ld [%o0], %o0`
	`sethi %hi(ptr), %o1`
	`or %lo(ptr), %o1`
	`ld [%l7 + %o1], %o1`
	`ld [%o1], %o1`
	`st %o0, [%o1]`

Function Calls

Programs use the `call` instruction to make direct function calls. Even when the code for a function resides in a shared object, the caller uses the same assembly language instruction sequence. A `call` instruction's destination is a PC-relative value that can reach any address in the 32-bit virtual space.

NOTE Although the assembly language is the same for absolute and position-independent code, the binary instruction sequences may differ. For example, when an executable file calls a shared object function, control passes from the original call, through an indirection sequence, to the desired destination. See ‘‘Procedure Linkage Table’’ in Chapter 5 for more information on the indirection sequence.

Figure 3-40: Direct Function Call, All Models

C	Assembly

`extern void function();`
`function();`
	`.global   function`
	`call      function`
	`nop`

Indirect function calls use the `jmpl` instruction. As explained elsewhere, register `%o7` holds the return address; so the compiler generates a `jmpl` instruction that follows this convention.

Figure 3-41: Absolute Indirect Function Call

C:

```
extern void (*ptr)();
extern void name();
ptr = name;




(*ptr)();
```

Assembly:

```
.global   ptr, name;

sethi     %hi(name), %o0
or        %lo(name), %o0
sethi     %hi(ptr), %g1
st        %o0, [%g1 + %lo(ptr)]

sethi     %hi(ptr), %g1
ld        [%g1 + %lo(ptr)], %g1
jmpl      %g1, %o7
nop
```

A global offset table holds absolute addresses for all required symbols, whether the symbols name objects or functions. Because the `call` instruction uses a PC-relative operand, a function can be called without needing its absolute address or a global offset table entry. Functions such as `name`, however, must have an entry, because their absolute address must be available.

Figure 3-42: Small Model Position-Independent Indirect Function Call

C	Assembly

`extern void (*ptr)();`
`extern void name();`
`ptr = name;`
	`.global ptr, name`
	
	`ld [%l7 + name], %o0`
	`ld [%l7 + ptr], %g1`
	`st %o0, [%g1]`

`(*ptr)();`	`ld [%l7 + ptr], %g1`
	`ld [%g1], %g1`
	`jmpl %g1, %o7`
	`nop`

Figure 3-43: Large Model Position-Independent Indirect Function Call

C	Assembly

`extern void (*ptr)();`
`extern void name();`
`ptr = name;`
	`.global ptr, name`
	
	`sethi %hi(name), %o0`
	`or %lo(name), %o0`
	`ld [%l7 + %o0], %o0`
	`sethi %hi(ptr), %g1`
	`or %lo(ptr), %g1`
	`ld [%l7 + %g1], %g1`
	`st %o0, [%g1]`

`(*ptr)();`	`sethi %hi(ptr), %g1`
	`or %lo(ptr), %g1`
	`ld [%l7 + %g1], %g1`
	`ld [%g1], %g1`
	`jmpl %g1, %o7`
	`nop`

Branching

Programs use branch instructions to control their execution flow. As defined by the architecture, branch instructions hold a PC-relative value with a 16 megabyte range, allowing a jump to locations up to 8 megabytes away in either direction.

Figure 3-44: Branch Instruction, All Models

C	Assembly

`label:`
`        . . .`
`        goto label;`
	`.L01:`
	`        . . .`
	`        ba      .L01`
	`        nop`

C `switch` statements provide multiway selection. When the `case` labels of a `switch` statement satisfy grouping constraints, the compiler implements the selection with an address table. The following examples use several simplifying conventions to hide irrelevant details:

- The selection expression resides in *local* register `%l0`;
- `case` label constants begin at zero;
- `case` labels, `default`, and the address table use assembly names `.Lcase`*i*, `.Ldef`, and `.Ltab`, respectively.

Address table entries for absolute code contain virtual addresses; the selection code extracts an entry's value and jumps to that address. Position-independent table entries hold offsets; the selection code computes a destination's absolute address.

Figure 3-45: Absolute `switch` Code

C:

```
switch (j)
{
case 0:
     . . .
case 2:
     . . .
case 3:
     . . .
default:
     . . .
}
```

Assembly:

```
        subcc   %l0, 4, %g0
        bgu     .Ldef
        sll     %l0, 2, %l0
        sethi   %hi(.Ltab), %o1
        or      %o1, %lo(.Ltab), %o1
        ld      [%l0 + %o1], %l0
        jmpl    %l0, %g0
        nop
.Ltab:  .word   .Lcase0
        .word   .Ldef
        .word   .Lcase2
        .word   .Lcase3
```

Figure 3-46: Position-Independent `switch` Code

C:

```
switch (j)
{
case 0:
     . . .
```

Assembly:

```
        subcc   %l0, 4, %g0
        bgu     .Ldef
        sll     %l0, 2, %l0
1:      call    2f
```

Figure 3-46: Position-Independent `switch` **Code** (continued)

```
case 2:
      . . .
case 3:
      . . .
default:
      . . .
}
```

```
          sethi   %hi(.Ltab - 1b), %g1
2:        or      %g1, %lo(.Ltab - 1b), %g1
          add     %l0, %g1, %l0
          ld      [%o7 + %l0], %l0
          jmpl    %o7 + %l0, %g0
          nop
.Ltab:    .word   .Lcase0 - 1b
          .word   .Ldef - 1b
          .word   .Lcase2 - 1b
          .word   .Lcase3 - 1b
```

C Stack Frame

Figure 3-47 shows the C stack frame organization. It conforms to the standard stack frame with designated roles for unspecified areas in the standard frame.

Figure 3-47: C Stack Frame

Base	Offset	Contents	Frame
`%fp`	–1	*y* words local space: automatic variables . . .	*High addresses*
`%fp`	$-4y$	other addressable objects	
`%sp`	$92+4x$	*x* words compiler scratch space: temporaries, register save area	
`%sp`	92	outgoing arguments 6, . . .	Current
		outgoing argument 5 . . .	
`%sp`	68	outgoing argument 0	
`%sp`	64	`struct`/`union` return pointer	
`%sp`	0	16-word window save area	*Low addresses*

A C stack frame doesn't normally change size during execution. The exception is dynamically allocated stack memory, discussed below. By convention, a function allocates automatic (local) variables in the top of its frame and references them as negative offsets from `%fp`. Its incoming arguments reside in the previous frame, referenced as positive offsets from `%fp`.

Variable Argument List

Previous sections describe the rules for passing arguments. Unfortunately, some otherwise portable C programs depend on the argument passing scheme, implicitly assuming that 1) all arguments reside on the stack, and 2) arguments appear in increasing order on the stack. Programs that make these assumptions never have been portable, but they have worked on many machines. They do *not* work on SPARC because the first 6 argument words reside in registers. Portable C programs should use the facilities defined in the header files <stdarg.h> or <varargs.h> to deal with variable argument lists (on SPARC and other machines as well).

When a function uses <stdarg.h> facilities, the compiler generates code in that function to move its register arguments to the stack's argument save area, thereafter treating them as regular stack objects. Argument registers are allocated in word order, meaning the stack locations for multi-word floating-point arguments may not be aligned properly. Thus a pointer to double might sometimes reference an unaligned object. Consequently, the compiler generates code to dereference ‘‘unknown’’ pointers one word at a time.

Figure 3-48: Argument Stack Positions

Call	Argument	Original	Stack
	word 0, 1.414	%i0	%fp+68
	word 1, 1.414	%i1	%fp+72
h(1.414, 1,	1	%i2	%fp+76
2.998e10,	word 0, 2.998e10	%i3	%fp+80
2.718);	word 1, 2.998e10	%i4	%fp+84
	word 0, 2.718	%i5	%fp+88
	word 1, 2.718	%fp+92	%fp+92

The save area for 1.414 is not doubleword aligned, because its offset, +68, is not a multiple of 8. Thus the compiler would load and store the value one word at a time. On the other hand, 2.718 resides at %fp+88, and the compiler can generate doubleword loads and stores. Alignments assume %fp and %sp hold doubleword addresses.

Allocating Stack Space Dynamically

Unlike some other languages, C does not need dynamic stack allocation *within* a stack frame. Frames are allocated dynamically on the program stack, depending on program execution, but individual stack frames can have static sizes. Nonetheless, the architecture supports dynamic allocation for those languages that require it, and the standard calling sequence and stack frame support it as well. Thus languages that need dynamic stack frame sizes can call C functions, and vice versa.

Figure 3-47 shows the layout of the C stack frame. The double line divides the area referenced with the frame pointer from the area referenced with the stack pointer. Dynamic space is allocated above the line as a downward growing heap whose size changes as required. Typical C functions have no space in the heap. All areas below the double line in the current frame have a known size to the compiler. Dynamic stack allocation thus takes the following steps.

1. Stack frames are doubleword aligned; dynamic allocation should preserve this property. Thus the program rounds (up) the desired byte count to a multiple of 8.
2. The program decreases the stack pointer by the rounded byte count, increasing its frame size. At this point, the ‘‘new’’ space resides just above the register save area at the bottom of the stack.
3. The program copies the ‘‘bottom half’’ of the stack frame down into the new space, opening the middle of the frame.

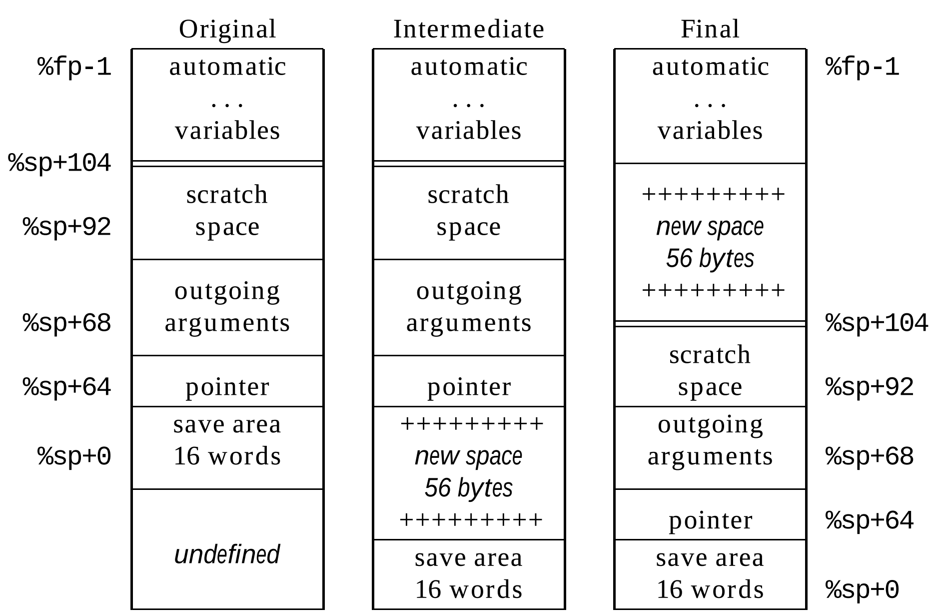
Even in the presence of signals, dynamic allocation is ‘‘safe.’’ If a signal interrupts allocation, one of three things can happen.

- The signal handler can return. The process then resumes the dynamic allocation from the point of interruption.
- The signal handler can execute a non-local goto, or `longjmp` [see `setjmp`(BA_LIB)]. This resets the process to a new context in a previous stack frame, automatically discarding the dynamic allocation.
- The process can terminate.

Regardless of when the signal arrives during dynamic allocation, the result is a consistent (though possibly dead) process.

To illustrate, assume a program wants to allocate 50 bytes; its current stack frame has 12 bytes of compiler scratch space and 24 bytes of outgoing arguments. The first step is rounding 50 to 56, making it a multiple of 8. Figure 3-49 shows how the stack frame changes.

Figure 3-49: Dynamic Stack Allocation



New space starts at `%sp+104`. As described, every dynamic allocation in *this* function will return a new area starting at `%sp+104`, leaving previous heap objects untouched (other functions would have different heap addresses). Consequently, the compiler should compute the absolute address for each area, avoiding relative references. Otherwise, future allocations in the same frame would destroy the heap’s integrity.

Existing stack objects reside at fixed offsets from the frame and stack pointers; stack heap allocation preserves those offsets. Objects relative to the frame pointer don't move. Objects relative to the stack pointer move, but their `%sp`-relative positions do not change. Accordingly, compilers arrange not to publicize the absolute address of any object in the bottom half of the stack frame (in a way that violates the scope rules). `%sp`-relative references stay valid after dynamic allocation, but absolute addresses do not.

No special code is needed to free dynamically allocated stack memory. The function return resets the stack pointer and removes the entire stack frame, including the heap, from the stack. Naturally, a program should not reference heap objects after they have gone out of scope.

4 OBJECT FILES

ELF Header

Machine Information

For file identification in e_ident, SPARC requires the following values.

Figure 4-1: SPARC Identification, e_ident

Position	Value
e_ident[EI_CLASS]	ELFCLASS32
e_ident[EI_DATA]	ELFDATA2MSB

Processor identification resides in the ELF header's e_machine member and must have the value 2, defined as the name EM_SPARC.

The ELF header's e_flags member holds bit flags associated with the file. SPARC defines no flags; so this member contains zero.

Sections

Special Sections

Various sections hold program and control information. Sections in the list below are used by the system and have the indicated types and attributes.

Figure 4-2: Special Sections

Name	Type	Attributes
.got	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.plt	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE + SHF_EXECINSTR
.sdata	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE

`.got` This section holds the global offset table. See ‘‘Coding Examples’’ in Chapter 3 and ‘‘Global Offset Table’’ in Chapter 5 for more information.

`.plt` This section holds the procedure linkage table. See ‘‘Procedure Linkage Table’’ in Chapter 5 for more information.

`.sdata` This section holds initialized data that contribute to the program’s memory image. The data are addressable by the short-form address convention.

Symbol Table

Symbol Values

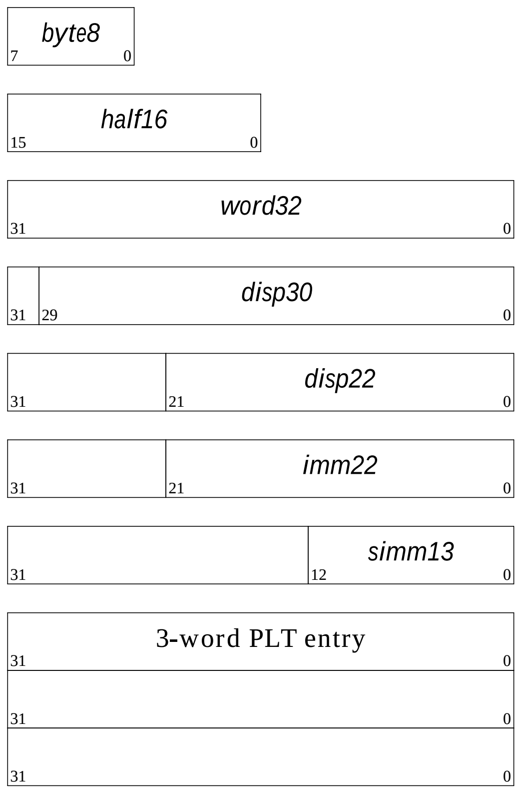
If an executable file contains a reference to a function defined in one of its associated shared objects, the symbol table section for that file will contain an entry for that symbol. The `st_shndx` member of that symbol table entry contains `SHN_UNDEF`. This informs the dynamic linker that the symbol definition for that function is not contained in the executable file itself. If that symbol has been allocated a procedure linkage table entry in the executable file, and the `st_value` member for that symbol table entry is non-zero, the value will contain the virtual address of the first instruction of that procedure linkage table entry. Otherwise, the `st_value` member contains zero. This procedure linkage table entry address is used by the dynamic linker in resolving references to the address of the function. See ‘‘Function Addresses’’ in Chapter 5 for details.

Relocation

Relocation Types

An overview of the instruction and data formats from *The SPARC Architecture Manual* makes relocation easier to understand. Relocation entries describe how to alter the following instruction and data fields (bit numbers appear in the lower box corners).

Figure 4-3: Relocatable Fields



Calculations below assume the actions are transforming a relocatable file into either an executable or a shared object file. Conceptually, the link editor merges one or more relocatable files to form the output. It first decides how to combine and locate the input files, then updates the symbol values, and finally performs the relocation. Relocations applied to executable or shared object files are similar and accomplish the same result. Descriptions below use the following notation.

A This means the addend used to compute the value of the relocatable field.

B This means the base address at which a shared object has been loaded into memory during execution. Generally, a shared object file is built with a 0 base virtual address, but the execution address will be different. See ''Program Header'' in the System V ABI for more information about the base address.

`G` This means the offset into the global offset table at which the address of the relocation entry's symbol will reside during execution. See ''Coding Examples'' in Chapter 3 and ''Global Offset Table'' in Chapter 5 for more information.

`L` This means the place (section offset or address) of the procedure linkage table entry for a symbol. A procedure linkage table entry redirects a function call to the proper destination. The link editor builds the initial procedure linkage table, and the dynamic linker modifies the entries during execution. See ''Procedure Linkage Table'' in Chapter 5 for more information.

`P` This means the place (section offset or address) of the storage unit being relocated (computed using `r_offset`).

`S` This means the value of the symbol whose index resides in the relocation entry.

Relocation entries apply to bytes (*byte8*), halfwords (*half16*), or words (the others). In any case, the `r_offset` value designates the offset or virtual address of the first byte of the affected storage unit. The relocation type specifies which bits to change and how to calculate their values. SPARC uses only `Elf32_Rela` relocation entries with explicit addends. Thus the `r_addend` member serves as the relocation addend.

NOTE Field names in the following table tell whether the relocation type checks for ''overflow.'' A calculated relocation value may be larger than the intended field, and a relocation type may verify (*V*) the value fits or truncate (*T*) the result. As an example, *V-simm13* means the the computed value may not have significant, non-zero bits outside the *simm13* field.

Figure 4-4: Relocation Types

Name	Value	Field	Calculation
`R_SPARC_NONE`	0	none	none
`R_SPARC_8`	1	*V-byte8*	`S + A`
`R_SPARC_16`	2	*V-half16*	`S + A`
`R_SPARC_32`	3	*V-word32*	`S + A`
`R_SPARC_DISP8`	4	*V-byte8*	`S + A - P`
`R_SPARC_DISP16`	5	*V-half16*	`S + A - P`
`R_SPARC_DISP32`	6	*V-word32*	`S + A - P`
`R_SPARC_WDISP30`	7	*V-disp30*	`(S + A - P) >> 2`
`R_SPARC_WDISP22`	8	*V-disp22*	`(S + A - P) >> 2`
`R_SPARC_HI22`	9	*T-imm22*	`(S + A) >> 10`
`R_SPARC_22`	10	*V-imm22*	`S + A`
`R_SPARC_13`	11	*V-simm13*	`S + A`
`R_SPARC_LO10`	12	*T-simm13*	`(S + A) & 0x3ff`
`R_SPARC_GOT10`	13	*T-simm13*	`G & 0x3ff`
`R_SPARC_GOT13`	14	*V-simm13*	`G`
`R_SPARC_GOT22`	15	*T-imm22*	`G >> 10`
`R_SPARC_PC10`	16	*T-simm13*	`(S + A - P) & 0x3ff`
`R_SPARC_PC22`	17	*V-disp22*	`(S + A - P) >> 10`
`R_SPARC_WPLT30`	18	*V-disp30*	`(L + A - P) >> 2`
`R_SPARC_COPY`	19	none	none
`R_SPARC_GLOB_DAT`	20	*V-word32*	`S + A`
`R_SPARC_JMP_SLOT`	21	none	see below

Figure 4-4: Relocation Types (continued)

R_SPARC_RELATIVE	22	*V-word32*	B + A
R_SPARC_UA32	23	*V-word32*	S + A

Some relocation types have semantics beyond simple calculation.

R_SPARC_GOT10
: This relocation type resembles R_SPARC_LO10, except it refers to the address of the symbol's global offset table entry and additionally instructs the link editor to build a global offset table.

R_SPARC_GOT13
: This relocation type resembles R_SPARC_13, except it refers to the address of the symbol's global offset table entry and additionally instructs the link editor to build a global offset table.

R_SPARC_GOT22
: This relocation type resembles R_SPARC_22, except it refers to the address of the symbol's global offset table entry and additionally instructs the link editor to build a global offset table.

R_SPARC_WPLT30
: This relocation type resembles R_SPARC_WDISP30, except it refers to the address of the symbol's procedure linkage table entry and additionally instructs the link editor to build a procedure linkage table.

R_SPARC_COPY
: The link editor creates this relocation type for dynamic linking. Its offset member refers to a location in a writable segment. The symbol table index specifies a symbol that should exist both in the current object file and in a shared object. During execution, the dynamic linker copies data associated with the shared object's symbol to the location specified by the offset.

R_SPARC_GLOB_DAT
: This relocation type resembles R_SPARC_32, except it is used to set a global offset table entry to the address of the specified symbol. The special relocation type allows one to determine the correspondence between symbols and global offset table entries.

R_SPARC_JMP_SLOT
: The link editor creates this relocation type for dynamic linking. Its offset member gives the location of a procedure linkage table entry. The dynamic linker modifies the procedure linkage table entry to transfer control to the designated symbol's address [see ''Procedure Linkage Table'' in Chapter 5].

R_SPARC_RELATIVE
: The link editor creates this relocation type for dynamic linking. Its offset member gives a location within a shared object that contains a value representing a relative address. The dynamic linker computes the corresponding virtual address by adding the virtual address at which the shared object was loaded to the relative address. Relocation entries for this type must specify 0 for the symbol table index.

R_SPARC_UA32
: This relocation type resembles R_SPARC_32, except it refers to an unaligned word. That is, the ''word'' to be relocated must be treated as four separate bytes with arbitrary alignment, not as a word aligned according to the architecture requirements.

5 PROGRAM LOADING AND DYNAMIC LINKING

Program Loading

As the system creates or augments a process image, it logically copies a file's segment to a virtual memory segment. When—and if—the system physically reads the file depends on the program's execution behavior, system load, etc. A process does not require a physical page unless it references the logical page during execution, and processes commonly leave many pages unreferenced. Therefore delaying physical reads frequently obviates them, improving system performance. To obtain this efficiency in practice, executable and shared object files must have segment images whose file offsets and virtual addresses are congruent, modulo the page size.

Virtual addresses and file offsets for SPARC segments are congruent modulo 64 K (`0x10000`) or larger powers of 2. Because 64 KB is the maximum page size, the files will be suitable for paging regardless of physical page size.

Figure 5-1: Executable File

File Offset	File	Virtual Address
`0`	ELF header	
	Program header table	
	Other information	

`0x100`	Text segment
	. . .
	`0x2be00` bytes
		`0x10100`
		
		`0x3beff`

`0x2bf00`	Data segment
	. . .
	`0x4e00` bytes
		`0x4bf00`
		
		`0x50cff`

`0x30d00`	Other information . . .	

Figure 5-2: Program Header Segments

Member	Text	Data
`p_type`	`PT_LOAD`	`PT_LOAD`
`p_offset`	`0x100`	`0x2bf00`
`p_vaddr`	`0x10100`	`0x4bf00`
`p_paddr`	unspecified	unspecified
`p_filesz`	`0x2be00`	`0x4e00`
`p_memsz`	`0x2be00`	`0x5e24`
`p_flags`	`PF_R + PF_X`	`PF_R + PF_W + PF_X`
`p_align`	`0x10000`	`0x10000`

Although the example's file offsets and virtual addresses are congruent modulo 64 K for both text and data, up to four file pages hold impure text or data (depending on page size and file system block size).

- The first text page contains the ELF header, the program header table, and other information.

- The last text page holds a copy of the beginning of data.
- The first data page has a copy of the end of text.
- The last data page may contain file information not relevant to the running process.

Logically, the system enforces the memory permissions as if each segment were complete and separate; segments' addresses are adjusted to ensure each logical page in the address space has a single set of permissions. In the example above, the region of the file holding the end of text and the beginning of data will be mapped twice: at one virtual address for text and at a different virtual address for data.

The end of the data segment requires special handling for uninitialized data, which the system defines to begin with zero values. Thus if a file's last data page includes information not in the logical memory page, the extraneous data must be set to zero, not the unknown contents of the executable file. ''Impurities'' in the other three pages are not logically part of the process image; whether the system expunges them is unspecified. The memory image for this program follows, assuming 4 KB (`0x1000`) pages.

Figure 5-3: Process Image Segments

Virtual Address	Contents	Segment
`0x10000`	*Header padding* `0x100` bytes	Text
`0x10100`	Text segment . . . `0x2be00` bytes	
`0x3bf00`	*Data padding* `0x100` bytes	
`0x4b000`	*Text padding* `0xf00` bytes	Data
`0x4bf00`	Data segment . . . `0x4e00` bytes	
`0x50d00`	Uninitialized data `0x1024` zero bytes	
`0x51d24`	*Page padding* `0x2dc` zero bytes	

One aspect of segment loading differs between executable files and shared objects. Executable file segments typically contain absolute code [see ''Coding Examples'' in Chapter 3]. To let the process execute correctly, the segments must reside at the virtual addresses used to build the executable file. Thus the system uses the `p_vaddr` values unchanged as virtual addresses.

On the other hand, shared object segments typically contain position-independent code. This lets a segment's virtual address change from one process to another, without invalidating execution behavior. Though the system chooses virtual addresses for individual processes, it maintains the segments' *relative positions*. Because position-independent code uses relative addressing between segments, the difference between virtual addresses in memory must match the difference between virtual addresses in the file. The following table shows possible shared object virtual address assignments for several processes, illustrating constant relative positioning. The table also illustrates the base address computations.

Figure 5-4: Example Shared Object Segment Addresses

Source	Text	Data	Base Address
File	0x200	0x2a400	0x0
Process 1	0xc0000200	0xc002a400	0xc0000000
Process 2	0xc0010200	0xc003a400	0xc0010000
Process 3	0xd0020200	0xd004a400	0xd0020000
Process 4	0xd0030200	0xd005a400	0xd0030000

Dynamic Linking

Dynamic Section

Dynamic section entries give information to the dynamic linker. Some of this information is processor-specific, including the interpretation of some entries in the dynamic structure.

DT_PLTGOT On SPARC, this entry's d_ptr member gives the address of the first entry in the procedure linkage table. As described below, the first entry is special, and the dynamic linker must know its address.

DT_JMP_REL As explained in the System V ABI, this entry is associated with a table of relocation entries for the procedure linkage table. For the SPARC processor, this entry is mandatory both for executable and shared object files. Moreover, the relocation table's entries must have a one-to-one correspondence with the procedure linkage table. See ''Procedure Linkage Table'' below for more information.

Global Offset Table

Position-independent code cannot, in general, contain absolute virtual addresses. Global offset tables hold absolute addresses in private data, thus making the addresses available without compromising the position-independence and sharability of a program's text. A program references its global offset table using position-independent addressing and extracts absolute values, thus redirecting position-independent references to absolute locations.

Initially, the global offset table holds information as required by its relocation entries [see ''Relocation'' in Chapter 4]. After the system creates memory segments for a loadable object file, the dynamic linker processes the relocation entries, some of which will be type R_SPARC_GLOB_DAT referring to the global offset table. The dynamic linker determines the associated symbol values, calculates their absolute addresses, and sets the appropriate memory table entries to the proper values. Although the absolute addresses are unknown when the link editor builds an object file, the dynamic linker knows the addresses of all memory segments and can thus calculate the absolute addresses of the symbols contained therein.

If a program requires direct access to the absolute address of a symbol, that symbol will have a global offset table entry. Because the executable file and shared objects have separate global offset tables, a symbol's address may appear in several tables. The dynamic linker processes all the global offset table relocations before giving control to any code in the process image, thus ensuring the absolute addresses are available during execution.

The table's entry zero is reserved to hold the address of the dynamic structure, referenced with the symbol _DYNAMIC. This allows a program, such as the dynamic linker, to find its own dynamic structure without having yet processed its relocation entries. This is especially important for the dynamic linker, because it must initialize itself without relying on other programs to relocate its memory image.

The system may choose different memory segment addresses for the same shared object in different programs; it may even choose different library addresses for different executions of the same program. Nonetheless, memory segments do not change addresses once the process image is established. As long as a process exists, its memory segments reside at fixed virtual addresses.

A global offset table's format and interpretation are processor-specific. For SPARC, the symbol _GLOBAL_OFFSET_TABLE_ may be used to access the table.

Figure 5-5: Global Offset Table

```
extern Elf32_Addr   _GLOBAL_OFFSET_TABLE_[];
```

The symbol `_GLOBAL_OFFSET_TABLE_` may reside in the middle of the `.got` section, allowing both negative and non-negative ‘‘subscripts’’ into the array of addresses.

Function Addresses

References to the address of a function from an executable file and the shared objects associated with it might not resolve to the same value. References from within shared objects will normally be resolved by the dynamic linker to the virtual address of the function itself. References from within the executable file to a function defined in a shared object will normally be resolved by the link editor to the address of the procedure linkage table entry for that function within the executable file.

To allow comparisons of function addresses to work as expected, if an executable file references a function defined in a shared object, the link editor will place the address of the procedure linkage table entry for that function in its associated symbol table entry. [See ‘‘Symbol Values’’ in Chapter 4]. The dynamic linker treats such symbol table entries specially. If the dynamic linker is searching for a symbol, and encounters a symbol table entry for that symbol in the executable file, it normally follows the rules below.

1. If the `st_shndx` member of the symbol table entry is not `SHN_UNDEF`, the dynamic linker has found a definition for the symbol and uses its `st_value` member as the symbol’s address.
2. If the `st_shndx` member is `SHN_UNDEF` and the symbol is of type `STT_FUNC` and the `st_value` member is not zero, the dynamic linker recognizes this entry as special and uses the `st_value` member as the symbol’s address.
3. Otherwise, the dynamic linker considers the symbol to be undefined within the executable file and continues processing.

Some relocations are associated with procedure linkage table entries. These entries are used for direct function calls rather than for references to function addresses. These relocations are not treated in the special way described above because the dynamic linker must not redirect procedure linkage table entries to point to themselves.

Procedure Linkage Table

Much as the global offset table redirects position-independent address calculations to absolute locations, the procedure linkage table redirects position-independent function calls to absolute locations. The link editor cannot resolve execution transfers (such as function calls) from one executable or shared object to another. Consequently, the link editor arranges to have the program transfer control to entries in the procedure linkage table. On SPARC, procedure linkage tables reside in private data. The dynamic linker determines the destinations’ absolute addresses and modifies the procedure linkage table’s memory image accordingly. The dynamic linker thus can redirect the entries without compromising the position-independence and sharability of the program’s text. Executable files and shared object files have separate procedure linkage tables.

The first four procedure linkage table entries are reserved. (The original contents of these entries are unspecified, despite the example below.) Each entry in the table occupies 3 words (12 bytes), and the last table entry must be followed by a `nop` instruction. As mentioned before, a relocation table is associated with the procedure linkage table. The `DT_JMP_REL` entry in the `_DYNAMIC` array gives the location of the first relocation entry. The relocation table's entries parallel the procedure linkage table in a one-to-one correspondence. That is, relocation table entry 0 applies to procedure linkage table entry 0, and so on. With the exception of the first four entries, the relocation type will be `R_SPARC_JMP_SLOT`, the relocation offset will specify the address of first byte of the associated procedure linkage table entry, and the symbol table index will reference the appropriate symbol.

To illustrate procedure linkage tables, the figure below shows four entries: two of the four initial reserved entries, a third to call `name1`, and a fourth to call `name2`. The example assumes the entry for `name2` is the table's last entry and shows the following `nop` instruction. The left column shows the instructions from the object file before dynamic linking. The right column demonstrates a possible way the dynamic linker might ‘‘fix’’ the procedure linkage table entries.

Figure 5-6: Procedure Linkage Table Example

Object File	Memory Segment

`.PLT0:`
`    unimp`
`    unimp`
`    unimp`
`.PLT1:`
`    unimp`
`    unimp`
`    unimp`
`    ...`
	`.PLT0:`
	`    save   %sp,-64,%sp`
	`    call   `*dynamic-linker*
	`    nop`
	`.PLT1:`
	`    .word  `*identification*
	`    unimp`
	`    unimp`
	`    ...`

`    ...`
`.PLT101:`
`    sethi  (.-.PLT0),%g1`
`    ba,a   .PLT0`
`    nop`
`.PLT102:`
`    sethi  (.-.PLT0),%g1`
`    ba,a   .PLT0`
`    nop`
	`    ...`
	`.PLT101:`
	`    sethi  (.-.PLT0),%g1`
	`    sethi  %hi(name1),%g1`
	`    jmpl   %g1+%lo(name1),%g0`
	`.PLT102:`
	`    sethi  (.-.PLT0),%g1`
	`    sethi  %hi(name2),%g1`
	`    jmpl   %g1+%lo(name2),%g0`

`    nop`	`    nop`

Following the steps below, the dynamic linker and the program ‘‘cooperate’’ to resolve symbolic references through the procedure linkage table. Again, the steps described below are for explanation only. The precise execution-time behavior of the dynamic linker is not specified.

1. When first creating the memory image of the program, the dynamic linker changes the initial procedure linkage table entries, making them transfer control to one of the dynamic linker's own routines: *dynamic-linker* above. It also stores a word of *identification* information in the second entry. When it receives control, it can examine this word to determine what object called it.

2. All other procedure linkage table entries initially transfer to the first entry, allowing the dynamic linker to can gain control at the first execution of each table entry. For illustration, assume the program calls `name1`, which transfers control to the label `.PLT101`.
3. The `sethi` instruction computes the distance between the current and the initial procedure linkage table entries, `.PLT101` and `.PLT0`, respectively. This value occupies the most significant 22 bits of the `%g1` register. In this example, `%g1` will contain `0x12f000` when the dynamic linker receives control.
4. Next, the `ba,a` instruction jumps to `.PLT0`, which then establishes a stack frame and calls the dynamic linker.
5. Using the *identification* value, the dynamic linker finds its data structures associated with the object in question, including the relocation table.
6. By shifting the `%g1` value and dividing by the size of each procedure linkage table entry, the dynamic linker computes the index of the relocation entry for `name1`. Relocation entry 101 will have type `R_SPARC_JMP_SLOT`, its offset will specify the address of `.PLT101`, and its symbol table index will reference `name1`.
7. Knowing this, the dynamic linker finds the symbol's ''real'' value, unwinds the stack, modifies the procedure linkage table entry, and transfers control to the desired destination.

Although the dynamic linker is not required to create the instruction sequences under the ''Memory Segment'' column, it might. Assuming it actually did, several points deserve further explanation.

- To make the code re-entrant, the procedure linkage table's instructions must be changed in a particular sequence. That is, if the dynamic linker is ''fixing'' a function's procedure linkage table entry and a signal arrives, the signal handling code must be able to call the original function with predictable (and correct) results.
- The dynamic linker must change two words to convert an entry; it can update each word atomically. Re-entrancy can be achieved by first overwriting the `nop` with the `jmpl` instruction, and then patching the `ba,a` to be `sethi`. If a re-entrant function call occurs between the two word updates, the `jmpl` will reside in the delay slot of the `ba,a` instruction, which annuls the delay instruction's effects. Consequently, the dynamic linker gains control a second time. Although both invocations of the dynamic linker modify the same procedure linkage table entry, their changes do not interfere with each other.
- The first `sethi` instruction of a procedure linkage table entry can fill the delay slot of the previous entry's `jmpl` instruction. Although the `sethi` changes the value of the `%g1` register, the previous contents can be safely discarded.
- After conversion, the last procedure linkage table entry (`.PLT102` above) needs a delay instruction for its `jmpl`. The required, trailing `nop` fills this delay slot.

The `LD_BIND_NOW` environment variable can change dynamic linking behavior. If its value is non-null, the dynamic linker evaluates procedure linkage table entries before transferring control to the program. That is, the dynamic linker processes relocation entries of type `R_SPARC_JMP_SLOT` during process initialization. Otherwise, the dynamic linker evaluates procedure linkage table entries lazily, delaying symbol resolution and relocation until the first execution of a table entry.

NOTE

Lazy binding generally improves overall application performance, because unused symbols do not incur the dynamic linking overhead. Nevertheless, two situations make lazy binding undesirable for some applications. First, the initial reference to a shared object function takes longer than subsequent calls, because the dynamic linker intercepts the call to resolve the symbol. Some applications cannot tolerate this unpredictability. Second, if an error occurs and the dynamic linker cannot resolve the symbol, the dynamic linker will terminate the program. Under lazy binding, this might occur at arbitrary times. Once again, some applications cannot tolerate this unpredictability. By turning off lazy binding, the dynamic linker forces the failure to occur during process initialization, before the application receives control.

Program Interpreter

There are two valid program interpreters for programs conforming to the SPARC ABI:

```
/usr/lib/ld.so.1    /usr/lib/libc.so.1
```

6 LIBRARIES

System Library

Support Routines

Besides operating system services, **libsys** contains the following processor-specific support routines.

Figure 6-1: `libsys` **Support Routines**

_Q_add	_Q_cmp	_Q_cmpe	_Q_div	_Q_dtoq
_Q_feq	_Q_fge	_Q_fgt	_Q_fle	_Q_flt
_Q_fne	_Q_itoq	_Q_mul	_Q_neg	_Q_qtod
_Q_qtoi	_Q_qtos	_Q_qtou	_Q_sqrt	_Q_stoq
_Q_sub	_Q_utoq	.div	__dtou	__ftou
.mul	.rem	.stret1	.stret2	.stret4
.stret8	.udiv	.umul	.urem	

Routines listed below employ the standard calling sequence that Chapter 3 describes in ''Function Calling Sequence.'' Descriptions are written from the *caller's* point of view, with respect to register usage and stack frame layout.

`long double _Q_add(long double a, long double b)`

This function corresponds to the SPARC `faddq` instruction. It returns `a` + `b` computed in quad-precision. The following aspects of exception handling mimic the `faddq` instruction: If any exceptions arise for which the corresponding TEM bits of the FSR are on, a `SIGFPE` will be generated, and the *aexc* field of the FSR will be unchanged. Otherwise any exceptions are OR'ed into the *aexc* field of the FSR.

`int _Q_cmp(long double a, long double b)`

This function compares `a` and `b` as quad-precision values and returns a value that indicates their relative ordering.

Relation	Value
`a` equal `b`	0
`a` less than `b`	1
`a` greater than `b`	2
`a` unordered with respect to `b`	3

The following aspects of exception handling mimic the `fcmpq` instruction: If any exceptions arise for which the corresponding TEM bits of the FSR are on, a `SIGFPE` will be generated, and the *aexc* field of the FSR will be unchanged. Otherwise any exceptions are OR'ed into the *aexc* field of the FSR. Upon return, the floating-point condition codes have unspecified values.

`int _Q_cmpe(long double a, long double b)`

This function compares `a` and `b` as quad-precision values and returns a value that indicates their relative ordering, using the same values as `_Q_cmp`. The following aspects of exception handling mimic the `fcmpeq` instruction: If any exceptions arise for which the corresponding TEM bits of the FSR are on, a `SIGFPE` will be generated, and the *aexc* field of the FSR will be unchanged. Otherwise any exceptions are OR'ed into the *aexc* field of the FSR. Upon return, the floating-point condition codes have unspecified values.

`long double _Q_div(long double a, long double b)`

This function corresponds to the SPARC `fdivq` instruction. It returns `a` / `b` computed in quad-precision. The following aspects of exception handling mimic the `fdivq` instruction: If any exceptions arise for which the corresponding TEM bits of the FSR are on, a `SIGFPE` will be generated, and the *aexc* field of the FSR will be unchanged. Otherwise any exceptions are OR'ed into the *aexc* field of the FSR.

`long double _Q_dtoq(double a)`

This function corresponds to the SPARC `fdtoq` instruction. It converts the double-precision input argument to quad-precision and returns the quad-precision value. The following aspects of exception handling mimic the `fdtoq` instruction: If any exceptions arise for which the corresponding TEM bits of the FSR are on, a `SIGFPE` will be generated, and the *aexc* field of the FSR will be unchanged. Otherwise any exceptions are OR'ed into the *aexc* field of the FSR.

`int _Q_feq(long double a, long double b)`

This function compares `a` and `b` as quad-precision values and returns a nonzero value if they are equal, zero otherwise. The following aspects of exception handling mimic the `fcmpq` instruction: If any exceptions arise for which the corresponding TEM bits of the FSR are on, a `SIGFPE` will be generated, and the *aexc* field of the FSR will be unchanged. Otherwise any exceptions are OR'ed into the *aexc* field of the FSR. Upon return, the floating-point condition codes have unspecified values.

`int _Q_fge(long double a, long double b)`

This function compares `a` and `b` as quad-precision values and returns a nonzero value if `a` is greater than or equal to `b`, zero otherwise. The following aspects of exception handling mimic the `fcmpeq` instruction: If any exceptions arise for which the corresponding TEM bits of the FSR are on, a `SIGFPE` will be generated, and the *aexc* field of the FSR will be unchanged. Otherwise any exceptions are OR'ed into the *aexc* field of the FSR. Upon return, the floating-point condition codes have unspecified values.

`int _Q_fgt(long double a, long double b)`

This function compares `a` and `b` as quad-precision values and returns a nonzero value if `a` is greater than `b`, zero otherwise. The following aspects of exception handling mimic the `fcmpeq` instruction: If any exceptions arise for which the corresponding TEM bits of the FSR are on, a `SIGFPE` will be generated, and the *aexc* field of the FSR will be unchanged. Otherwise any exceptions are OR'ed into the *aexc* field of the FSR. Upon return, the floating-point condition codes have unspecified values.

`int _Q_fle(long double a, long double b)`

This function compares `a` and `b` as quad-precision values and returns a nonzero value if `a` is less than or equal to `b`, zero otherwise. The following aspects of exception handling mimic the `fcmpeq` instruction: If any exceptions arise for which the corresponding TEM bits of the FSR are on, a `SIGFPE` will be generated, and the *aexc* field of the FSR will be unchanged. Otherwise any exceptions are OR'ed into the *aexc* field of the FSR. Upon return, the floating-point condition codes have unspecified values.

`int _Q_flt(long double a, long double b)`

This function compares `a` and `b` as quad-precision values and returns a nonzero value if `a` is less than `b`, zero otherwise. The following aspects of exception handling mimic the `fcmpeq` instruction: If any exceptions arise for which the corresponding TEM bits of the FSR are on, a `SIGFPE` will be generated, and the *aexc* field of the FSR will be unchanged. Otherwise any exceptions are OR'ed into the *aexc* field of the FSR. Upon return, the floating-point condition codes have unspecified values.

`int _Q_fne(long double a, long double b)`

This function compares `a` and `b` as quad-precision values and returns a nonzero value if they are unordered or not equal, zero otherwise. The following aspects of exception handling mimic the `fcmpq` instruction: If any exceptions arise for which the corresponding TEM bits of the FSR are on, a `SIGFPE` will be generated, and the *aexc* field of the FSR will be unchanged. Otherwise any exceptions are OR'ed into the *aexc* field of the FSR. Upon return, the floating-point condition codes have unspecified values.

`long double _Q_itoq(int a)`

This function corresponds to the SPARC `fitoq` instruction. It converts the integer input argument to quad-precision and returns the quad-precision value. `_Q_itoq` raises no exceptions.

`long double _Q_mul(long double a, long double b)`

This function corresponds to the SPARC `fmulq` instruction. It returns $a \times b$ computed in quad-precision. The following aspects of exception handling mimic the `fmulq` instruction: If any exceptions arise for which the corresponding TEM bits of the FSR are on, a `SIGFPE` will be generated, and the *aexc* field of the FSR will be unchanged. Otherwise any exceptions are OR'ed into the *aexc* field of the FSR.

`long double _Q_neg(long double a)`

This function corresponds to the SPARC `fnegs` instruction. It returns $-a$ computed in quad-precision. `_Q_neg` raises no exceptions.

`double _Q_qtod(long double a)`

This function corresponds to the SPARC `fqtod` instruction. It converts the quad-precision input argument to double-precision and returns the double-precision value. The following aspects of exception handling mimic the `fqtod` instruction: If any exceptions arise for which the corresponding TEM bits of the FSR are on, a `SIGFPE` will be generated, and the *aexc* field of the FSR will be unchanged. Otherwise any exceptions are OR'ed into the *aexc* field of the FSR.

`int _Q_qtoi(long double a)`

This function corresponds to the SPARC `fqtoi` instruction. It converts the quad-precision input argument to a signed 32-bit integer and returns the integer value. The following aspects of exception handling mimic the `fqtoi` instruction: If any exceptions arise for which the corresponding TEM bits of the FSR are on, a `SIGFPE` will be generated, and the *aexc* field of the FSR will be unchanged. Otherwise any exceptions are OR'ed into the *aexc* field of the FSR.

`float _Q_qtos(long double a)`

This function corresponds to the SPARC `fqtos` instruction. It converts the quad-precision input argument to single-precision and returns the single-precision value. The following aspects of exception handling mimic the `fqtos` instruction: If any exceptions arise for which the corresponding TEM bits of the FSR are on, a `SIGFPE` will be generated, and the *aexc* field of the FSR will be unchanged. Otherwise any exceptions are OR'ed into the *aexc* field of the FSR.

`unsigned int _Q_qtou(long double a)`

This function converts the quad-precision input argument to an unsigned integer (discarding any fractional part) and returns the unsigned integer value. `_Q_qtou` raises exceptions as follows.

If $0 \leq$ a $< 2^{32}$, the operation is successful. If a is a whole number, no exceptions are raised. If a is not a whole number, the inexact exception is raised.

Otherwise, the value returned by _Q_qtou is unspecified, and the invalid exception is raised.

If any exceptions arise for which the corresponding TEM bits of the FSR are on, a SIGFPE will be generated, and the *aexc* field of the FSR will be unchanged. Otherwise, any exceptions are OR'ed into the *aexc* field of the FSR. (Note that _Q_qtou is present for the convenience of compilers and has no direct counterpart in the SPARC instruction set.)

`long double _Q_sqrt(long double a)`

This function corresponds to the SPARC fsqrtq instruction. It returns the square root of its argument, computed in quad-precision. The following aspects of exception handling mimic the fsqrtq instruction: If any exceptions arise for which the corresponding TEM bits of the FSR are on, a SIGFPE will be generated, and the *aexc* field of the FSR will be unchanged. Otherwise any exceptions are OR'ed into the *aexc* field of the FSR.

`long double _Q_stoq(float a)`

This function corresponds to the SPARC fstoq instruction. It converts the single-precision input argument to quad-precision, and returns the quad-precision value. The following aspects of exception handling mimic the fstoq instruction: If any exceptions arise for which the corresponding TEM bits of the FSR are on, a SIGFPE will be generated, and the *aexc* field of the FSR will be unchanged. Otherwise any exceptions are OR'ed into the *aexc* field of the FSR.

`long double _Q_sub(long double a, long double b)`

This function corresponds to the SPARC fsubq instruction. It returns a – b computed in quad-precision. The following aspects of exception handling mimic the fsubq instruction: If any exceptions arise for which the corresponding TEM bits of the FSR are on, a SIGFPE will be generated, and the *aexc* field of the FSR will be unchanged. Otherwise any exceptions are OR'ed into the *aexc* field of the FSR.

`long double _Q_utoq(unsigned int a)`

This function converts the unsigned integer value in its argument to quad-precision, and returns the quad-precision value. _Q_utoq raises no exceptions. (Note that _Q_utoq is present for the convenience of compilers and has no direct counterpart in the SPARC instruction set.)

`int .div(int a, int b)`

This function computes a / b with signed integer division, leaving the result in the caller's %o0 register. Truncation is toward zero, regardless of the operands' signs. If the divisor (b) is zero, the function generates a software trap 2, with the consequences specified in ''Operating System Interface'' of Chapter 3. Upon return, the integer condition codes and registers %o1 through %o5 have unspecified values.

`unsigned int __dtou(double a)`

This function converts the double-precision input argument to an unsigned integer (discarding any fractional part) and returns the unsigned integer value. __dtou raises exceptions as follows.

If $0 \leq$ `a` $< 2^{32}$, the operation is successful. If `a` is a whole number, no exceptions are raised. If `a` is not a whole number, the inexact exception is raised.

Otherwise, the value returned by `__dtou` is unspecified, and the invalid exception is raised.

If any exceptions arise for which the corresponding TEM bits of the FSR are on, a `SIGFPE` will be generated, and the *aexc* field of the FSR will be unchanged. Otherwise, any exceptions are OR'ed into the *aexc* field of the FSR. (Note that `__dtou` is present for the convenience of compilers and has no direct counterpart in the SPARC instruction set.)

`unsigned int __ftou(float a)`

This function converts the single-precision input argument to an unsigned integer (discarding any fractional part) and returns the unsigned integer value. `__ftou` raises exceptions as follows.

If $0 \leq$ `a` $< 2^{32}$, the operation is successful. If `a` is a whole number, no exceptions are raised. If `a` is not a whole number, the inexact exception code is raised.

Otherwise, the value returned by `__ftou` is unspecified, and the invalid exception is raised.

If any exceptions arise for which the corresponding TEM bits of the FSR are on, a `SIGFPE` will be generated, and the *aexc* field of the FSR will be unchanged. Otherwise, any exceptions are OR'ed into the *aexc* field of the FSR. (Note that `__ftou` is present for the convenience of compilers and has no direct counterpart in the SPARC instruction set.)

`int .mul(int a, int b)`

This function computes `a` $\times$ `b` with signed integer multiplication. When `.mul` returns, the caller's register `%o0` contains the least significant 32 bits of the 64-bit result; register `%o1` holds the most significant 32 bits of the result. Upon return, the integer condition codes and registers `%o2` through `%o5` have unspecified values.

`int .rem(int a, int b)`

This function computes the signed integer remainder of `a` / `b`, leaving the result in the caller's `%o0` register. The remainder has the same sign as the dividend. If the divisor (`b`) is zero, the function generates a software trap 2, with the consequences specified in ‘‘Operating System Interface’’ of Chapter 3. Upon return, the integer condition codes and registers `%o1` through `%o5` have unspecified values.

`.stret1, .stret2, .stret4, .stret8`

Although these entry points work with the standard calling sequence, they should not be *called*. Instead, a function that returns a structure, union, or quad-precision value may transfer control to one of these entry points, which in turn will copy the result structure's value, restore the original caller's context, and return control to the original caller. Descriptions are written from the current stack frame's point of view, with respect to register usage and stack frame layout. That is, the routines run in the stack frame of the function that was called to return the structure, union, or quad-precision value; they use the following interface.

`%o0` This register holds the address of the object to be returned. That is, the value of this object will be copied into the space supplied by the original caller.

`%o1` This register holds the size, in bytes, of the object the called function intends to return to its caller.

`%fp+64` The word residing at `%fp+64` holds the address of the destination object supplied by the caller. That is, the object to which `%o0` points will be copied to the space addressed by `%fp+64`.

`%i7` As the standard calling sequence specifies, this register holds the address of the original `call` instruction.

`%i7+8` Again following the calling sequence for functions that return structures, unions, or quad-precision values, the word at this address should be an `unimp` instruction. The least significant 12 bits of the instruction hold the least significant 12 bits of the size of the object expected by the caller.

The entry points perform the following steps.

1. They verify the word at `%i7+8` is an `unimp` instruction. If it is not, they restore the caller's context and return control to the word addressed by `%i7+8`.
2. If `%i7+8` is an `unimp` instruction, they compare the low order 12 bits of the instruction to the low order 12 bits of `%o1`. If the actual and expected sizes do not match, the entry points restore the caller's context and return control to the word addressed by `%i7+8`.
3. If the low order 12 bits of the sizes match, they copy `%o1` bytes from the object addressed by `%o0` to the object addressed by the word residing at `%fp+64`.
4. After copying the return object, they set `%i0` to the address of the destination object, restore the caller's context, and return control to the word addressed by `%i7+12`.

Upon return, the integer condition codes and the called function's registers `%i1` through `%i5` have unspecified values. Moreover, the value of `%i0` is unspecified too, unless the program successfully copies the return object to its destination.

Four entry points exist to handle the four possible alignment constraints for structured objects. That is, `.stret1`, `.stret2`, `.stret4`, and `.stret8` should be used when both the source and the destination are aligned on at least a byte, halfword, word, or doubleword boundary, respectively. If either the source or the destination object has insufficient alignment for the entry point used, the program has undefined behavior. For example, if the address of the caller's destination object is odd and the called function uses `.stret2` to return its value, the program behavior is undefined.

`unsigned .udiv(unsigned a, unsigned b)`

This function computes `a` / `b` with unsigned integer division, leaving the result in the caller's `%o0` register. If the divisor (`b`) is zero, the function generates a software trap 2, with the consequences specified in ‘‘Operating System Interface’’ of Chapter 3. Upon return, the integer condition codes and registers `%o1` through `%o5` have unspecified values.

`unsigned .umul(unsigned a, unsigned b)`

This function computes `a` × `b` with unsigned integer multiplication. When `.umul` returns, the caller's register `%o0` contains the least significant 32 bits of the 64-bit result; register `%o1` holds the most significant 32 bits of the result. Upon return, the integer condition codes and registers `%o2` through `%o5` have unspecified values.

```
unsigned .urem(unsigned a, unsigned b)
```

This function computes the unsigned integer remainder of `a` / `b`, leaving the result in the caller's `%o0` register. If the divisor (`b`) is zero, the function generates a software trap 2, with the consequences specified in ''Operating System Interface'' of Chapter 3. Upon return, the integer condition codes and registers `%o1` through `%o5` have unspecified values.

Global Data Symbols

The `libsys` library requires that some global external data objects be defined for the routines to work properly. In addition to the corresponding data symbols listed in the System V ABI, the following symbols must be provided in the system library on all ABI-conforming systems implemented with the SPARC processor architecture. Declarations for the data objects listed below can be found in the Data Definitions section of this chapter or immediately following the table.

Figure 6-2: `libsys`**, Global External Data Symbols**

`__huge_val`

Application Constraints

As described above, `libsys` provides symbols for applications. In a few cases, however, an application is obliged to provide symbols for the library. In addition to the application-provided symbols listed in this section of the System V ABI, conforming applications on the SPARC processor architecture are also required to provide the following symbols.

`extern _end;`
This symbol refers neither to a routine nor to a location with interesting contents. Instead, its address must correspond to the beginning of a program's dynamic allocation area, called the heap. Typically, the heap begins immediately after the data segment of the program's executable file.

`extern const int _lib_version;`
This variable's value specifies the compilation and execution mode for the program. If the value is zero, the program wants to preserve the semantics of older (pre-ANSI) C, where conflicts exist with ANSI. Otherwise, the value is non-zero, and the program wants ANSI C semantics.

System Data Interfaces

Data Definitions

This section contains standard header files that describe system data. These files are referred to by their names in angle brackets: <*name.h*> and <sys/*name.h*>. Included in these headers are macro definitions and data definitions.

The data objects described in this section are part of the interface between an ABI-conforming application and the underlying ABI-conforming system where it will run. While an ABI-conforming system must provide these interfaces, it is not required to contain the actual header files referenced here. Programmers should observe that the sources of the structures defined in these headers are defined in SVID.

ANSI C serves as the ABI reference programming language, and data definitions are specificed in ANSI C format. The C language is used here as a convenient notation. Using a C language description of these data objects does *not* preclude their use by other programming languages.

Figure 6-3: `<assert.h>`

```
extern void __assert(const char *, const char *, int);
#define assert(EX) (void)((EX)||(__assert(#EX, __FILE__, __LINE__), 0))
```

Figure 6-4: `<ctype.h>`

```
#define _U      01
#define _L      02
#define _N      04
#define _S      010
#define _P      020
#define _C      040
#define _B      0100
#define _X      0200

extern unsigned char     _ _ctype[];

#define isalpha(c)      ((__ctype+1)[c]&(_U|_L))
#define isupper(c)      ((__ctype+1)[c]&_U)
#define islower(c)      ((__ctype+1)[c]&_L)
#define isdigit(c)      ((__ctype+1)[c]&_N)
#define isxdigit(c)     ((__ctype+1)[c]&_X)
#define isalnum(c)      ((__ctype+1)[c]&(_U|_L|_N))
#define isspace(c)      ((__ctype+1)[c]&_S)
#define ispunct(c)      ((__ctype+1)[c]&_P)
#define isprint(c)      ((__ctype+1)[c]&(_P|_U|_L|_N|_B))
#define isgraph(c)      ((__ctype+1)[c]&(_P|_U|_L|_N))
#define iscntrl(c)      ((__ctype+1)[c]&_C)
#define isascii(c)      (!((c)&~0177))
#define _toupper(c)     ((__ctype+258)[c])
#define _tolower(c)     ((__ctype+258)[c])
#define toascii(c)      ((c)&0177)
```

CAUTION The data definitions in ctype.h are moved to Level 2 as of Jan. 1 1993. In order to correctly function in an internationalized environment, applications are encouraged to use the functions in libc/libsys instead.

Figure 6-5: <dirent.h>

```
typedef struct {
        int     dd_fd;
        int     dd_loc;
        int     dd_size;
        char    *dd_buf;
} DIR;

struct dirent {
        ino_t           d_ino;
        off_t           d_off;
        unsigned short  d_reclen;
        char            d_name[1];
};

#define rewinddir( dirp )       seekdir( dirp, 0L )
```

Figure 6-6: <errno.h>

```
extern int errno;

#define EPERM           1
#define ENOENT          2
#define ESRCH           3
#define EINTR           4
#define EIO             5
#define ENXIO           6
#define E2BIG           7
#define ENOEXEC         8
#define EBADF           9
#define ECHILD          10
#define EAGAIN          11
#define ENOMEM          12
#define EACCES          13
#define EFAULT          14
#define ENOTBLK         15
#define EBUSY           16
#define EEXIST          17
#define EXDEV           18
#define ENODEV          19
#define ENOTDIR         20
#define EISDIR          21
#define EINVAL          22
#define ENFILE          23
#define EMFILE          24
#define ENOTTY          25
#define ETXTBSY         26
```

(continued on next page)

Figure 6-6: <errno.h> (continued)

```
#define EFBIG           27
#define ENOSPC          28
#define ESPIPE          29
#define EROFS           30
#define EMLINK          31
#define EPIPE           32
#define EDOM            33
#define ERANGE          34
#define ENOMSG          35
#define EIDRM           36
#define ECHRNG          37
#define EL2NSYNC        38
#define EL3HLT          39
#define EL3RST          40
#define ELNRNG          41
#define EUNATCH         42
#define ENOCSI          43
#define EL2HLT          44
#define EDEADLK         45
#define ENOLCK          46
#define ENOSTR          60
#define ENODATA         61
#define ETIME           62
#define ENOSR           63
#define ENONET          64
#define ENOPKG          65
#define EREMOTE         66
#define ENOLINK         67
#define EADV            68
#define ESRMNT          69
#define ECOMM           70
#define EPROTO          71
#define EMULTIHOP               74
#define EBADMSG         77
#define ENAMETOOLONG    78
#define EOVERFLOW               79
#define ENOTUNIQ                80
#define EBADFD          81
#define EREMCHG         82
#define ENOSYS          89
#define ELOOP           90
#define ERESTART        91
#define ESTRPIPE        92
#define ENOTEMPTY               93
#define EUSERS          94
#define ECONNABORTED    130
#define ECONNRESET              131
#define ECONNREFUSED    146
#define ESTALE          151
```

Figure 6-7: `<fcntl.h>`

```
#define O_RDONLY 0
#define O_WRONLY 1
#define O_RDWR   2
#define O_APPEND        010
#define O_SYNC   020
#define O_NONBLOCK      0200
#define O_CREAT  00400
#define O_TRUNC  01000
#define O_EXCL   02000
#define O_NOCTTY        04000

#define F_DUPFD  0
#define F_GETFD  1
#define F_SETFD  2
#define F_GETFL  3
#define F_SETFL  4
#define F_GETLK  14
#define F_SETLK  6
#define F_SETLKW        7

#define FD_CLOEXEC      1
#define O_ACCMODE       3

typedef struct flock {
        short   l_type;
        short   l_whence;
        off_t   l_start;
        off_t   l_len;
        long    l_sysid;
        pid_t   l_pid;
        long    pad[4];
} flock_t;

#define F_RDLCK  01
#define F_WRLCK  02
#define F_UNLCK  03
```

Figure 6-8: `<float.h>`

```
extern int __flt_rounds;
#define FLT_ROUNDS      __flt_rounds
```

Figure 6-9: `<fmtmsg.h>`

```
#define MM_NULL      0L

#define MM_HARD      0x00000001L
#define MM_SOFT      0x00000002L
#define MM_FIRM      0x00000004L
#define MM_RECOVER   0x00000100L
#define MM_NRECOV    0x00000200L
#define MM_APPL      0x00000008L
#define MM_UTIL      0x00000010L
#define MM_OPSYS     0x00000020L
#define MM_PRINT     0x00000040L
#define MM_CONSOLE   0x00000080L

#define MM_NOSEV     0
#define MM_HALT      1
#define MM_ERROR     2
#define MM_WARNING   3
#define MM_INFO      4

#define MM_NULLLBL   ((char *) 0)
#define MM_NULLSEV   MM_NOSEV
#define MM_NULLMC    0L
#define MM_NULLTXT   ((char *) 0)
#define MM_NULLACT   ((char *) 0)
#define MM_NULLTAG   ((char *) 0)

#define MM_NOTOK     -1
#define MM_OK        0x00
#define MM_NOMSG     0x01
#define MM_NOCON     0x04
```

Figure 6-10: `<ftw.h>`

```
#define FTW_PHYS        01
#define FTW_MOUNT       02
#define FTW_CHDIR       04
#define FTW_DEPTH       010

#define FTW_F   0
#define FTW_D   1
#define FTW_DNR 2
#define FTW_NS  3
#define FTW_SL  4
#define FTW_DP  6

struct FTW
{
        int     quit;
        int     base;
        int     level;
};
```

Figure 6-11: `<grp.h>`

```
struct  group {
        char    *gr_name;
        char    *gr_passwd;
        gid_t   gr_gid;
        char    **gr_mem;
};
```

Figure 6-12: `<sys/ipc.h>`

```
struct ipc_perm {
        uid_t           uid;
        gid_t           gid;
        uid_t           cuid;
        gid_t           cgid;
        mode_t          mode;
        unsigned long   seq;
        key_t           key;
        long            pad[4];
};

#define IPC_CREAT               0001000
#define IPC_EXCL                0002000
#define IPC_NOWAIT              0004000

#define IPC_PRIVATE     (key_t)0

#define IPC_RMID                10
#define IPC_SET         11
#define IPC_STAT                12
```

Figure 6-13: `<langinfo.h>`

```
#define DAY_1       1
#define DAY_2       2
#define DAY_3       3
#define DAY_4       4
#define DAY_5       5
#define DAY_6       6
#define DAY_7       7

#define ABDAY_1     8
#define ABDAY_2     9
#define ABDAY_3     10
#define ABDAY_4     11
#define ABDAY_5     12
#define ABDAY_6     13
#define ABDAY_7     14

#define MON_1       15
#define MON_2       16
#define MON_3       17
#define MON_4       18
#define MON_5       19
#define MON_6       20
#define MON_7       21
#define MON_8       22
#define MON_9       23
```

(continued on next page)

Figure 6-13: `<langinfo.h>` (continued)

```
#define MON_10      24
#define MON_11      25
#define MON_12      26

#define ABMON_1     27
#define ABMON_2     28
#define ABMON_3     29
#define ABMON_4     30
#define ABMON_5     31
#define ABMON_6     32
#define ABMON_7     33
#define ABMON_8     34
#define ABMON_9     35
#define ABMON_10    36
#define ABMON_11    37
#define ABMON_12    38

#define RADIXCHAR   39
#define THOUSEP     40
#define YESSTR      41
#define NOSTR       42
#define CRNCYSTR    43

#define D_T_FMT     44
#define D_FMT       45
#define T_FMT       46
#define AM_STR      47
#define PM_STR      48
```

Figure 6-14: `<limits.h>`

```
#define MB_LEN_MAX                5

#define ARG_MAX           *
#define CHILD_MAX                 *
#define MAX_CANON                 *
#define NGROUPS_MAX       *
#define LINK_MAX          *
#define NAME_MAX          *
#define OPEN_MAX          *
#define PASS_MAX          *
#define PATH_MAX          *
#define PIPE_BUF          *
#define MAX_INPUT                 *

/* starred values vary and should be retrieved using sysconf() or pathconf() */

#define NL_ARGMAX                 9
#define NL_LANGMAX                14
#define NL_MSGMAX                 32767
#define NL_NMAX           1
#define NL_SETMAX                 255
#define NL_TEXTMAX                255
#define NZERO             20
#define TMP_MAX           17576
#define FCHR_MAX          1048576
```

Figure 6-15: `<locale.h>`

```
struct lconv {
        char    *decimal_point;
        char    *thousands_sep;
        char    *grouping;
        char    *int_curr_symbol;
        char    *currency_symbol;
        char    *mon_decimal_point;
        char    *mon_thousands_sep;
        char    *mon_grouping;
        char    *positive_sign;
        char    *negative_sign;
        char    int_frac_digits;
        char    frac_digits;
        char    p_cs_precedes;
        char    p_sep_by_space;
        char    n_cs_precedes;
        char    n_sep_by_space;
        char    p_sign_posn;
        char    n_sign_posn;
};

#define LC_CTYPE        0
#define LC_NUMERIC              1
#define LC_TIME         2
#define LC_COLLATE              3
#define LC_MONETARY     4
#define LC_MESSAGES     5
#define LC_ALL          6
#define NULL            0
```

Figure 6-16: `<math.h>`

```
typedef union _h_val {
        unsigned long   i[2];
        double          d;
} _h_val;

extern const _h_val      __huge_val;
#define HUGE_VAL __huge_val.d
```

Figure 6-17: `<sys/mman.h>`

```
#define PROT_READ                0x1
#define PROT_WRITE               0x2
#define PROT_EXEC                0x4
#define PROT_NONE                0x0

#define MAP_SHARED               1
#define MAP_PRIVATE      2
#define MAP_FIXED                0x10

#define MS_SYNC          0x0
#define MS_ASYNC         0x1
#define MS_INVALIDATE    0x2

#define PROC_TEXT        (PROT_EXEC | PROT_READ)
#define PROC_DATA        (PROT_READ | PROT_WRITE | PROT_EXEC)

#define SHARED           0x10
#define PRIVATE          0x20

#define MC_SYNC          1
#define MC_LOCK          2
#define MC_UNLOCK        3
#define MC_LOCKAS        5
#define MC_UNLOCKAS      6

#define MCL_CURRENT      0x1
#define MCL_FUTURE       0x2
```

Figure 6-18: `<sys/mount.h>`

```
#define MS_RDONLY                0x01
#define MS_DATA          0x04
#define MS_NOSUID                0x10
#define MS_REMOUNT               0x20
```

Figure 6-19: `<sys/msg.h>`

```
struct msqid_ds {
        struct ipc_perm msg_perm;
        struct msg      *msg_first;
        struct msg      *msg_last;
        unsigned long   msg_cbytes;
        unsigned long   msg_qnum;
        unsigned long   msg_qbytes;
        pid_t           msg_lspid;
        pid_t           msg_lrpid;
        time_t          msg_stime;
        long            msg_pad1;
        time_t          msg_rtime;
        long            msg_pad2;
        time_t          msg_ctime;
        long            msg_pad3;
        long            msg_pad4[4];
};

#define MSG_NOERROR     010000
```

Figure 6-20: <netconfig.h>

```
struct  netconfig {
        char            *nc_netid;
        unsigned long   nc_semantics;
        unsigned long   nc_flag;
        char            *nc_protofmly;
        char            *nc_proto;
        char            *nc_device;
        unsigned long   nc_nlookups;
        char            **nc_lookups;
        unsigned long   nc_unused[8];
};
#define NC_TPI_CLTS      1
#define NC_TPI_COTS      2
#define NC_TPI_COTS_ORD  3
#define NC_TPI_RAW       4
#define NC_NOFLAG        00
#define NC_VISIBLE       01
#define NC_NOPROTOFMLY   "-"
#define NC_LOOPBACK      "loopback"
#define NC_INET          "inet"
#define NC_IMPLINK       "implink"
#define NC_PUP           "pup"
#define NC_CHAOS         "chaos"
#define NC_NS            "ns"
#define NC_NBS           "nbs"
#define NC_ECMA          "ecma"
#define NC_DATAKIT       "datakit"
#define NC_CCITT         "ccitt"
#define NC_SNA           "sna"
#define NC_DECNET        "decnet"
#define NC_DLI           "dli"
#define NC_LAT           "lat"
#define NC_HYLINK        "hylink"
#define NC_APPLETALK     "appletalk"
#define NC_NIT           "nit"
#define NC_IEEE802       "ieee802"
#define NC_OSI           "osi"
#define NC_X25           "x25"
#define NC_OSINET        "osinet"
#define NC_GOSIP         "gosip"
#define NC_NOPROTO       "-"
#define NC_TCP           "tcp"
#define NC_UDP           "udp"
#define NC_ICMP          "icmp"
```

Figure 6-21: `<netdir.h>`

```
struct nd_addrlist {
        int             n_cnt;
        struct netbuf   *n_addrs;
};

struct nd_hostservlist {
        int  h_cnt;
        struct nd_hostserv  *h_hostservs;
};

struct nd_hostserv {
        char  *h_host;
        char  *h_serv;
};

#define ND_BADARG            -2
#define ND_NOMEM             -1
#define ND_OK                0
#define ND_NOHOST            1
#define ND_NOSERV            2
#define ND_NOSYM             3
#define ND_OPEN              4
#define ND_ACCESS            5
#define ND_UKNWN             6
#define ND_NOCTRL            7
#define ND_FAILCTRL          8
#define ND_SYSTEM            9
#define ND_HOSTSERV          0
#define ND_HOSTSERVLIST      1
#define ND_ADDR              2
#define ND_ADDRLIST          3

#define                      HOST_SELF"\\1"
#define                      HOST_ANY"\\2"
#define                      HOST_BROADCAST"\\3"

#define ND_SET_BROADCAST     1
#define ND_SET_RESERVEDPORT  2
#define ND_CHECK_RESERVEDPORT3
#define ND_MERGEADDR         4
```

Figure 6-22: `<nl_types.h>`

```
#define NL_SETD         1

typedef short nl_item ;
typedef void *nl_catd;
```

Figure 6-23: `<sys/param.h>`

```
#define CANBSIZ         256
#define HZ              100

#define NGROUPS_UMIN    0

#define MAXPATHLEN              1024
#define MAXSYMLINKS     20
#define MAXNAMELEN              256

#define NADDR           13

#define PIPE_MAX                5120

#define NBBY            8
#define NBPSCTR         512
```

Figure 6-24: `<poll.h>`

```
struct pollfd {
        int     fd;
        short   events;
        short   revents;
};

#define POLLIN       0x0001
#define POLLPRI      0x0002
#define POLLOUT      0x0004
#define POLLRDNORM   0x0040
#define POLLWRNORM   POLLOUT
#define POLLRDBAND   0x0080
#define POLLWRBAND   0x0100
#define POLLNORM     POLLRDNORM

#define POLLERR      0x0008
#define POLLHUP      0x0010
#define POLLNVAL     0x0020
```

Figure 6-25: `<sys/procset.h>`

```
#define P_INITPID      1
#define P_INITUID      0
#define P_INITPGID     0

typedef long id_t;

typedef enum idtype {
        P_PID,
        P_PPID,
        P_PGID,
        P_SID,
        P_CID,
        P_UID,
        P_GID,
        P_ALL
} idtype_t;

typedef enum idop {
        POP_DIFF,
        POP_AND,
        POP_OR,
        POP_XOR
} idop_t;

typedef struct procset {
        idop_t          p_op;
        idtype_t        p_lidtype;
        id_t            p_lid;
        idtype_t        p_ridtype;
        id_t            p_rid;
} procset_t;

#define P_MYID          (-1)
```

Figure 6-26: `<pwd.h>`

```
struct passwd {
        char    *pw_name;
        char    *pw_passwd;
        uid_t   pw_uid;
        gid_t   pw_gid;
        char    *pw_age;
        char    *pw_comment;
        char    *pw_gecos;
        char    *pw_dir;
        char    *pw_shell;
};
```

Figure 6-27: `<sys/resource.h>`

```
#define RLIMIT_CPU      0
#define RLIMIT_FSIZE    1
#define RLIMIT_DATA     2
#define RLIMIT_STACK    3
#define RLIMIT_CORE     4
#define RLIMIT_NOFILE   5
#define RLIMIT_VMEM     6
#define RLIMIT_AS               RLIMIT_VMEM
#define RLIM_INFINITY   0x7fffffff

typedef unsigned long rlim_t;

struct rlimit {
        rlim_t  rlim_cur;
        rlim_t  rlim_max;
};
```

Figure 6-28: `<rpc.h>`

```
#define MAX_AUTH_BYTES   400
#define MAXNETNAMELEN    255
#define HEXKEYBYTES      48

enum auth_stat {
        AUTH_OK=0,
        AUTH_BADCRED=1,
        AUTH_REJECTEDCRED=2,
        AUTH_BADVERF=3,
        AUTH_REJECTEDVERF=4,
        AUTH_TOOWEAK=5,
        AUTH_INVALIDRESP=6,
        AUTH_FAILED=7
};

union des_block {
        struct {
          unsigned long high;
          unsigned long low;
        } key;
        char c[8];
};
struct opaque_auth {
  int   oa_flavor;
  char *oa_base;
  unsigned int  oa_length;
};

typedef struct {
        struct  opaque_auth  ah_cred;
        struct  opaque_auth  ah_verf;
        union   des_block  ah_key;
        struct auth_ops {
          void  (*ah_nextverf)();
          int   (*ah_marshal)();
          int   (*ah_validate)();
          int   (*ah_refresh)();
          void  (*ah_destroy)();
        } *ah_ops;
        char *ah_private;
} AUTH;
```

(continued on next page)

Figure 6-28: <rpc.h> (continued)

```
struct authsys_parms {
        unsigned long  aup_time;
        char  *aup_machname;
        uid_t  aup_uid;
        gid_t  aup_gid;
        unsigned int  aup_len;
        gid_t  *aup_gids;
};
extern struct opaque_auth _null_auth;

#define AUTH_NONE      0
#define AUTH_NULL      0
#define AUTH_SYS       1
#define AUTH_UNIX      AUTH_SYS
#define AUTH_SHORT     2
#define AUTH_DES       3

enum clnt_stat {
        RPC_SUCCESS=0,
        RPC_CANTENCODEARGS=1,
        RPC_CANTDECODERES=2,
        RPC_CANTSEND=3,
        RPC_CANTRECV=4,
        RPC_TIMEDOUT=5,
        RPC_INTR=18,
        RPC_UDERROR=23,
        RPC_VERSMISMATCH=6,
        RPC_AUTHERROR=7,
        RPC_PROGUNAVAIL=8,
        RPC_PROGVERSMISMATCH=9,
        RPC_PROCUNAVAIL=10,
        RPC_CANTDECODEARGS=11,
        RPC_SYSTEMERROR=12,
        RPC_UNKNOWNHOST=13,
        RPC_UNKNOWNPROTO=17,
        RPC_UNKNOWNADDR=19,
        RPC_NOBROADCAST=21,
        RPC_RPCBFAILURE=14,
        RPC_PROGNOTREGISTERED=15,
        RPC_N2AXLATEFAILURE=22,
        RPC_TLIERROR=20,
        RPC_FAILED=16
};
```

(continued on next page)

Figure 6-28: <rpc.h> (continued)

```
#define RPC_PMAPFAILURE RPC_RPCBFAILURE
#define RPC_ANYSOCK      -1
#define RPC_ANYFD        RPC_ANYSOCK

struct rpc_err {
        enum clnt_stat re_status;
        union {
                struct {
                  int errno;
                  int t_errno;
                } RE_err;
                enum auth_stat RE_why;
                struct {
                  unsigned long low;
                  unsigned long high;
                } RE_vers;
                struct {
                  long s1;
                  long s2;
                } RE_lb;
        } ru;
};

struct rpc_createerr {
  enum clnt_stat cf_stat;
  struct rpc_err cf_error;
};

typedef struct {
        AUTH    *cl_auth;
        struct clnt_ops {
          enum clnt_stat (*cl_call)();
          void (*cl_abort)();
          void (*cl_geterr)();
          int (*cl_freeres)();
          void (*cl_destroy)();
          int (*cl_control)();
        } *cl_ops;
        char    *cl_private;
        char    *cl_netid;
        char    *cl_tp;
} CLIENT;

#define FEEDBACK_REXMIT1                1
#define FEEDBACK_OK             2

#define CLSET_TIMEOUT           1
#define CLGET_TIMEOUT           2
#define CLGET_SERVER_ADDR               3
#define CLGET_FD                6
#define CLGET_SVC_ADDR          7
#define CLSET_FD_CLOSE          8
#define CLSET_FD_NCLOSE         9
```

(continued on next page)

Figure 6-28: <rpc.h> (continued)

```
#define CLSET_RETRY_TIMEOUT             4
#define CLGET_RETRY_TIMEOUT             5
extern struct
rpc_createerr rpc_createerr;

enum xprt_stat {
  XPRT_DIED,
  XPRT_MOREREQS,
  XPRT_IDLE
};

typedef struct {
        int  xp_fd;
        unsigned short xp_port;
        struct xp_ops {
            int (*xp_recv)();
            enum xprt_stat (*xp_stat)();
            int (*xp_getargs)();
            int (*xp_reply)();
            int (*xp_freeargs)();
            void (*xp_destroy)();
        } *xp_ops;
        int     xp_addrlen;
        char    *xp_tp;
        char    *xp_netid;
        struct netbuf xp_ltaddr;
        struct netbuf xp_rtaddr;
        char    xp_raddr[16];
        struct opaque_auth xp_verf;
        char    *xp_p1;
        char    *xp_p2;
        char    *xp_p3;
} SVCXPRT;

struct svc_req {
        unsigned long rq_prog;
        unsigned long rq_vers;
        unsigned long rq_proc;
        struct opaque_auth rq_cred;
        char     *rq_clntcred;
        SVCXPRT  *rq_xprt;
};

typedef struct fd_set {
        long fds_bits[32];
} fd_set;
extern fd_set svc_fdset;

enum msg_type {
        CALL=0,
        REPLY=1
};
```

(continued on next page)

Figure 6-28: <rpc.h> (continued)

```
enum reply_stat {
        MSG_ACCEPTED=0,
        MSG_DENIED=1
};

enum accept_stat {
        SUCCESS=0,
        PROG_UNAVAIL=1,
        PROG_MISMATCH=2,
        PROC_UNAVAIL=3,
        GARBAGE_ARGS=4,
        SYSTEM_ERR=5
};

enum reject_stat {
        RPC_MISMATCH=0,
        AUTH_ERROR=1
};

struct accepted_reply {
        struct opaque_auth ar_verf;
        enum accept_stat ar_stat;
        union {
         struct {
          unsigned long low;
          unsigned long high;
         } AR_versions;
         struct {
          char *where;
          xdrproc_t proc;
         } AR_results;
        } ru;
};

struct rejected_reply {
        enum reject_stat rj_stat;
        union {
         struct {
          unsigned long low;
          unsigned long high;
         } RJ_versions;
         enum auth_stat RJ_why;
        } ru;
};

struct reply_body {
        enum reply_stat rp_stat;
        union {
          struct accepted_reply RP_ar;
          struct rejected_reply RP_dr;
        } ru;
};
```

(continued on next page)

Figure 6-28: <rpc.h> (continued)

```
struct call_body {
        unsigned long cb_rpcvers;
        unsigned long cb_prog;
        unsigned long cb_vers;
        unsigned long cb_proc;
        struct opaque_auth cb_cred;
        struct opaque_auth cb_verf;
};

struct rpc_msg {
        unsigned long rm_xid;
        enum msg_type rm_direction;
        union {
          struct call_body RM_cmb;
          struct reply_body RM_rmb;
        } ru;
};

struct rpcb {
        unsigned long r_prog;
        unsigned long r_vers;
        char *r_netid;
        char *r_addr;
        char *r_owner;
};

struct rpcblist {
  struct rpcb rpcb_map;
  struct rpcblist *rpcb_next;
};

enum xdr_op {
  XDR_ENCODE=0,
  XDR_DECODE=1,
  XDR_FREE=2
};

struct xdr_discrim {
  int   value;
  xdrproc_t proc;
};

enum authdes_namekind {
  ADN_FULLNAME,
  ADN_NICKNAME
};

struct authdes_fullname {
  char *name;
  union des_block key;
  unsigned long window;
};
```

(continued on next page)

Figure 6-28: <rpc.h> (continued)

```
struct authdes_cred {
  enum authdes_namekind adc_namekind;
  struct authdes_fullname adc_fullname;
  unsigned long adc_nickname;
};
typedef struct {
        enum xdr_op     x_op;
        struct xdr_ops {
          int   (*x_getlong)();
          int   (*x_putlong)();
          int   (*x_getbytes)();
          int   (*x_putbytes)();
          unsigned int  (*x_getpostn)();
          int  (*x_setpostn)();
          long *(*x_inline)();
          void  (*x_destroy)();
        } *x_ops;
        char *x_public;
        char *x_private;
        char *x_base;
        int   x_handy;
} XDR;

typedef int (*xdrproc_t)()
#define NULL_xdrproc_t ((xdrproc_t)0)
#define auth_destroy(auth) \
        ((*((auth)->ah_ops->ah_destroy))(auth))
#define clnt_call(rh, proc, xargs, argsp, xres, resp, secs) \
        ((*(rh)->cl_ops->cl_call)(rh, proc, xargs, argsp, xres, resp, secs))
#define clnt_freeres(rh,xres,resp) \
        ((*(rh)->cl_ops->cl_freeres)(rh,xres,resp))
#define clnt_geterr(rh, errp) \
        ((*(rh)->cl_ops->cl_geterr)(rh, errp))
#define clnt_control(cl, rq, in) \
        ((*(cl)->cl_ops->cl_control)(cl, rq, in))
#define clnt_destroy(rh) \
        ((*(rh)->cl_ops->cl_destroy)(rh))
#define svc_destroy(xprt) \
        (*(xprt)->xp_ops->xp_destroy)(xprt)
#define svc_freeargs(xprt, xargs, argsp) \
        (*(xprt)->xp_ops->xp_freeargs)((xprt), (xargs), (argsp))
#define svc_getargs(xprt, xargs, argsp) \
        (*(xprt)->xp_ops->xp_getargs)((xprt), (xargs), (argsp))
#define svc_getrpccaller(x) \
        (&(x)->xp_rtaddr)
#define xdr_getpos(xdrs) \
        (*(xdrs)->x_ops->x_getpostn)(xdrs)
#define xdr_setpos(xdrs, pos) \
        (*(xdrs)->x_ops->x_setpostn)(xdrs, pos)
#define xdr_inline(xdrs, len) \
        (*(xdrs)->x_ops->x_inline)(xdrs, len)
#define xdr_destroy(xdrs) \
        (*(xdrs)->x_ops->x_destroy)(xdrs)
```

(continued on next page)

Figure 6-28: <rpc.h> (continued)

Figure 6-29: <search.h>

```
typedef struct entry { char *key; void *data; } ENTRY;
typedef enum { FIND, ENTER } ACTION;
typedef enum { preorder, postorder, endorder, leaf } VISIT;
```

Figure 6-30: `<sys/sem.h>`

```
#define SEM_UNDO        010000

#define GETNCNT         3
#define GETPID          4
#define GETVAL          5
#define GETALL          6
#define GETZCNT         7
#define SETVAL          8
#define SETALL          9

struct semid_ds {
        struct ipc_perm sem_perm;
        struct sem      *sem_base;
        unsigned short  sem_nsems;
        time_t          sem_otime;
        long            sem_pad1;
        time_t          sem_ctime;
        long            sem_pad2;
        long            sem_pad3[4];
};

struct sem {
        unsigned short  semval;
        pid_t           sempid;
        unsigned short  semncnt;
        unsigned short  semzcnt;
};

struct sembuf {
        unsigned short  sem_num;
        short           sem_op;
        short           sem_flg;
};
```

Figure 6-31: `<setjmp.h>`

```
#define _JBLEN        12
#define _SIGJBLEN     19
typedef int jmp_buf[_JBLEN];
typedef int sigjmp_buf[_SIGJBLEN];
```

Figure 6-32: `<sys/shm.h>`

```
struct shmid_ds {
        struct ipc_perm shm_perm;
        int             shm_segsz;
        struct anon_map *shm_amp;
        unsigned short  shm_lkcnt;
        pid_t           shm_lpid;
        pid_t           shm_cpid;
        unsigned long   shm_nattch;
        unsigned long   shm_cnattch;
        time_t          shm_atime;
        long            shm_pad1;
        time_t          shm_dtime;
        long            shm_pad2;
        time_t          shm_ctime;
        long            shm_pad3;
        long            shm_pad4[4];
};

#define SHM_RDONLY      010000
#define SHM_RND         020000
```

Figure 6-33: `<signal.h>`

```
#define SIGHUP          1
#define SIGINT          2
#define SIGQUIT         3
#define SIGILL          4
#define SIGTRAP         5
#define SIGABRT         6
#define SIGEMT          7
#define SIGFPE          8
#define SIGKILL         9
#define SIGBUS          10
#define SIGSEGV         11
#define SIGSYS          12
#define SIGPIPE         13
#define SIGALRM         14
#define SIGTERM         15
#define SIGUSR1         16
#define SIGUSR2         17
#define SIGCHLD         18
#define SIGPWR          19
#define SIGWINCH        20
#define SIGURG          21
#define SIGPOLL         22
#define SIGSTOP         23
```

(continued on next page)

Figure 6-33: `<signal.h>` (continued)

```
#define SIGTSTP         24
#define SIGCONT         25
#define SIGTTIN         26
#define SIGTTOU         27
#define SIGXCPU         30
#define SIGXFSZ         31
#define SIG_BLOCK               1
#define SIG_UNBLOCK     2
#define SIG_SETMASK     3
#define SIG_ERR         (void(*)())-1
#define SIG_IGN         (void(*)())1
#define SIG_HOLD        (void(*)())2
#define SIG_DFL         (void(*)())0

#define SS_ONSTACK              0x00000001
#define SS_DISABLE              0x00000002

struct sigaltstack {
        char    *ss_sp;
        int     ss_size;
        int     ss_flags;
};
typedef struct sigaltstack stack_t;
typedef struct { unsigned long sigbits[4]; } sigset_t;
struct sigaction {
        int             sa_flags;
        sigdisp_t               sa_disp;
        sigset_t        sa_mask;
        int             sa_resv[2];
};

#define SA_ONSTACK      0x00000001
#define SA_RESETHAND    0x00000002
#define SA_RESTART      0x00000004
#define SA_SIGINFO      0x00000008
#define SA_NOCLDWAIT    0x00010000
#define SA_NOCLDSTOP    0x00020000
```

Figure 6-34: `<sys/siginfo.h>`

```
#define ILL_ILLOPC      1
#define ILL_ILLOPN      2
#define ILL_ILLADR      3
#define ILL_ILLTRP      4
#define ILL_PRVOPC      5
#define ILL_PRVREG      6
#define ILL_COPROC      7
#define ILL_BADSTK      8
#define FPE_INTDIV      1
#define FPE_INTOVF      2
#define FPE_FLTDIV      3
#define FPE_FLTOVF      4
#define FPE_FLTUND      5
#define FPE_FLTRES      6
#define FPE_FLTINV      7
#define FPE_FLTSUB      8
#define SEGV_MAPERR     1
#define SEGV_ACCERR     2
#define BUS_ADRALN      1
#define BUS_ADRERR      2
#define BUS_OBJERR      3
#define TRAP_BRKPT      1
#define TRAP_TRACE      2
#define CLD_EXITED      1
#define CLD_KILLED      2
#define CLD_DUMPED      3
#define CLD_TRAPPED     4
#define CLD_STOPPED     5
#define CLD_CONTINUED   6
#define POLL_IN         1
#define POLL_OUT        2
#define POLL_MSG        3
#define POLL_ERR        4
#define POLL_PRI        5
#define POLL_HUP        6
#define SI_MAXSZ        128
#define SI_PAD  ((SI_MAXSZ/sizeof(int)) - 3)

typedef struct siginfo {
        int  si_signo;
        int  si_code;
        int  si_errno;
        union {
         int    _pad[SI_PAD];
         struct {
                pid_t  _pid;
                union {
                struct { uid_t _uid; } _kill;
                struct {
                        clock_t _utime;
                        int     _status;
                        clock_t _stime;
                } _cld;
```

(continued on next page)

Figure 6-34: <sys/siginfo.h> (continued)

```
          } _pdata;
          } _proc;
          struct { char *_addr; } _fault;
          struct {
                 int     _fd;
                 long    _band;
          } _file;
         } _data;
} siginfo_t;

#define si_pid       _data._proc._pid
#define si_uid       _data._proc._pdata._kill._uid
#define si_addr      _data._fault._addr
#define si_status    _data._proc._pdata._cld._status
#define si_band      _data._file._band
```

Figure 6-35: <sys/stat.h>

```
#define _ST_FSTYPSZ      16

struct  stat {
        dev_t        st_dev;
        longst_pad1[3];
        ino_t        st_ino;
        mode_t       st_mode;
        nlink_t      st_nlink;
        uid_t        st_uid;
        gid_t        st_gid;
        dev_t        st_rdev;
        long         st_pad2[2];
        off_t        st_size;
        long         st_pad3;
        timestruc_t  st_atim;
        timestruc_t  st_mtim;
        timestruc_t  st_ctim;
        long         st_blksize;
        long         st_blocks;
        char         st_fstype[_ST_FSTYPSZ];
        long         st_pad4[8];
};

#define st_atime     st_atim.tv_sec
#define st_mtime     st_mtim.tv_sec
#define st_ctime     st_ctim.tv_sec
#define S_IFMT       0xF000
#define S_IFIFO      0x1000
```

(continued on next page)

Figure 6-35: `<sys/stat.h>` (continued)

```
#define S_IFCHR      0x2000
#define S_IFDIR      0x4000
#define S_IFBLK      0x6000
#define S_IFREG      0x8000
#define S_IFLNK      0xA000
#define S_ISUID      04000
#define S_ISGID      02000
#define S_ISVTX      01000
#define S_IRWXU      00700
#define S_IRUSR      00400
#define S_IWUSR      00200
#define S_IXUSR      00100
#define S_IRWXG      00070
#define S_IRGRP      00040
#define S_IWGRP      00020
#define S_IXGRP      00010
#define S_IRWXO      00007
#define S_IROTH      00004
#define S_IWOTH      00002
#define S_IXOTH      00001

#define S_ISFIFO(mode)((mode&S_IFMT) == S_IFIFO)
#define S_ISCHR(mode)((mode&S_IFMT) == S_IFCHR)
#define S_ISDIR(mode)((mode&S_IFMT) == S_IFDIR)
#define S_ISBLK(mode)((mode&S_IFMT) == S_IFBLK)
#define S_ISREG(mode)((mode&S_IFMT) == S_IFREG)
```

Figure 6-36: `<sys/statvfs.h>`

```
#define FSTYPSZ 16

typedef struct statvfs {
        unsigned long f_bsize;
        unsigned long f_frsize;
        unsigned long f_blocks;
        unsigned long f_bfree;
        unsigned long f_bavail;
        unsigned long f_files;
        unsigned long f_ffree;
        unsigned long f_favail;
        unsigned long f_fsid;
        char          f_basetype[FSTYPSZ];
        unsigned long f_flag;
        unsigned long f_namemax;
        char          f_fstr[32];
        unsigned long f_filler[16];
} statvfs_t;

#define ST_RDONLY       0x01
#define ST_NOSUID       0x02
```

Figure 6-37: `<stdarg.h>`

```
#define _VA_LIST void *

typedef _VA_LIST  va_list;

#define va_start(list, name) (void) (list = (va_list) &__builtin_va_alist)
#define va_arg(list, mode) ((mode *)__builtin_va_arg_incr((mode *)list))[0]

extern void va_end(va_list);

#define va_end(list) (void)0
```

Figure 6-38: `<stddef.h>`

```
#define NULL             0
typedef int              ptrdiff_t;
typedef unsigned int     size_t;
typedef long             wchar_t;
```

Figure 6-39: `<stdio.h>`

```
typedef unsigned int size_t;
typedef long         fpos_t;

#define NULL         0
#define BUFSIZ       1024
#define _IOFBF       0000
#define _IOLBF       0100
#define _IONBF       0004
#define _IOEOF       0020
#define _IOERR       0040
#define EOF          (-1)
#define FOPEN_MAX    20
#define FILENAME_MAX 1024

#define stdin        (&__iob[0])
#define stdout       (&__iob[1])
#define stderr       (&__iob[2])

#define clearerr(p)  ((void)((p)->_flag &= ~(_IOERR | _IOEOF))) †
#define feof(p)      ((p)->_flag & _IOEOF)
#define ferror(p)    ((p)->_flag & _IOERR)
#define fileno(p)    (p)->_file  †


#define L_ctermid    9
#define L_cuserid    9
#define P_tmpdir     "/var/tmp/"
#define L_tmpnam     25

typedef struct {
             int_cnt;
             unsigned char*_ptr;
             unsigned char*_base;
             unsigned char_flag;
             unsigned char_file;
} FILE;

extern FILE  __iob[FOPEN_MAX];
```

† These macros definitions are moved to Level 2 in this release. †† The `_file` member of the `FILE` struct is moved to Level 2 as of Jan. 1 1993.

NOTE The macros `clearerr`, and `fileno` will be removed as a source interface in a future release supporting multi-processing. This will have no effect on binary portability.

CAUTION The constant `_NFILE` has been removed. It should still appear in stdio.h, but may be removed in a future version of the header file. Applications may not be able to depend on `fopen()` failing on an attempt to open more than `_NFILE` files.

Figure 6-40: `<stdlib.h>`

```
typedef struct {
        int       quot;
        int       rem;
} div_t;

typedef struct {
        long    quot;
        long    rem;
} ldiv_t;

typedef unsigned int    size_t;

#define NULL            0
#define EXIT_FAILURE    1
#define EXIT_SUCCESS    0
#define RAND_MAX        32767

extern unsigned char   __ctype[];
#define MB_CUR_MAX     __ctype[520]
```

Figure 6-41: `<stropts.h>`

```
#define SNDZERO     0x001
#define RNORM       0x000
#define RMSGD       0x001
#define RMSGN       0x002
#define RMODEMASK   0x003
#define RPROTDAT    0x004
#define RPROTDIS    0x008
#define RPROTNORM   0x010

#define FLUSHR      0x01
#define FLUSHW      0x02
#define FLUSHRW     0x03

#define S_INPUT     0x0001
#define S_HIPRI     0x0002
#define S_OUTPUT    0x0004
#define S_MSG       0x0008
#define S_ERROR     0x0010
#define S_HANGUP    0x0020
```

(continued on next page)

Figure 6-41: `<stropts.h>` (continued)

```
#define S_RDNORM     0x0040
#define S_WRNORM     S_OUTPUT
#define S_RDBAND     0x0080
#define S_WRBAND     0x0100
#define S_BANDURG    0x0200

#define RS_HIPRI     1
#define MSG_HIPRI    0x01
#define MSG_ANY      0x02
#define MSG_BAND     0x04

#define MORECTL      1
#define MOREDATA     2

#define MUXID_ALL    (-1)
#define STR          ('S'<<8)
#define I_NREAD      (STR|01)
#define I_PUSH       (STR|02)
#define I_POP        (STR|03)
#define I_LOOK       (STR|04)
#define I_FLUSH      (STR|05)
#define I_SRDOPT     (STR|06)
#define I_GRDOPT     (STR|07)
#define I_STR        (STR|010)
#define I_SETSIG     (STR|011)
#define I_GETSIG     (STR|012)
#define I_FIND       (STR|013)
#define I_LINK       (STR|014)
#define I_UNLINK     (STR|015)
#define I_PEEK       (STR|017)
#define I_FDINSERT   (STR|020)
#define I_SENDFD     (STR|021)
#define I_RECVFD     (STR|016)
#define I_SWROPT     (STR|023)
#define I_GWROPT     (STR|024)
#define I_LIST       (STR|025)
#define I_PLINK      (STR|026)
#define I_PUNLINK    (STR|027)
#define I_FLUSHBAND  (STR|034)
#define I_CKBAND     (STR|035)
#define I_GETBAND    (STR|036)
#define I_ATMARK     (STR|037)
#define I_SETCLTIME  (STR|040)
#define I_GETCLTIME  (STR|041)
#define I_CANPUT     (STR|042)

struct strioctl {
        int     ic_cmd;
        int     ic_timout;
        int     ic_lec;
        char    *ic_dp;
};

struct strbuf {
        int     maxlen;
```

(continued on next page)

Figure 6-41: `<stropts.h>` (continued)

```
        int     len;
        char    *buf;
};

struct strpeek {
        struct strbuf ctlbuf;
        struct strbuf databuf;
        long          flags;
};

struct strfdinsert {
        struct strbuf ctlbuf;
        struct strbuf databuf;
        long          flags;
        int           fildes;
        int           offset;
};

struct strrecvfd {
        int     fd;
        uid_t   uid;
        gid_t   gid;
        char    fill[8];
};

#define FMNAMESZ 8

struct str_mlist {
        char l_name[FMNAMESZ+1];
};

struct str_list {
        int               sl_nmods;
        struct str_mlist *sl_modlist;
};

#define ANYMARK 0x01
#define LASTMARK 0x02

struct bandinfo {
        unsigned charbi_pri;
        int     bi_flag;
};
```

Figure 6-42: `<termios.h>`

```
#define NCC              8
#define NCCS             19
#define CTRL(c)          ((c)&037)

#define IBSHIFT          16
#define _POSIX_VDISABLE  0

typedef unsigned long    tcflag_t;
typedef unsigned char    cc_t;
typedef unsigned long    speed_t;

#define VINTR    0
#define VQUIT    1
#define VERASE   2
#define VKILL    3
#define VEOF     4
#define VEOL     5
#define VEOL2    6
#define VMIN     4
#define VTIME    5
#define VSWTCH   7
#define VSTART   8
#define VSTOP    9
#define VSUSP    10
#define VDSUSP   11
#define VREPRINT         12
#define VDISCARD         13
#define VWERASE  14
#define VLNEXT   15
#define CNUL     0
#define CDEL     0177
#define CESC     '\\'
#define CINTR    0177
#define CQUIT    034
#define CERASE   '#'
#define CKILL    '@'
#define CEOT     04
#define CEOL     0
#define CEOL2    0
#define CEOF     04
#define CSTART   021
#define CSTOP    023
#define CSWTCH   032
#define CNSWTCH  0
#define CSUSP    CTRL('z')
#define CDSUSP   CTRL('y')
#define CRPRNT   CTRL('r')
#define CFLUSH   CTRL('o')
#define CWERASE  CTRL('w')
#define CLNEXT   CTRL('v')

#define IGNBRK   0000001
#define BRKINT   0000002
```

(continued on next page)

Figure 6-42: `<termios.h>` (continued)

```
#define IGNPAR  0000004
#define PARMRK  0000010
#define INPCK   0000020
#define ISTRIP  0000040
#define INLCR   0000100
#define IGNCR   0000200
#define ICRNL   0000400
#define IUCLC   0001000
#define IXON    0002000
#define IXANY   0004000
#define IXOFF   0010000
#define IMAXBEL 0020000
#define OPOST   0000001
#define OLCUC   0000002
#define ONLCR   0000004
#define OCRNL   0000010
#define ONOCR   0000020
#define ONLRET  0000040
#define OFILL   0000100
#define OFDEL   0000200
#define NLDLY   0000400
#define NL0     0
#define NL1     0000400
#define CRDLY   0003000
#define CR0     0
#define CR1     0001000
#define CR2     0002000
#define CR3     0003000
#define TABDLY  0014000
#define TAB0    0
#define TAB1    0004000
#define TAB2    0010000
#define TAB3    0014000
#define XTABS   TAB3
#define BSDLY   0020000
#define BS0     0
#define BS1     0020000
#define VTDLY   0040000
#define VT0     0
#define VT1     0040000
#define FFDLY   0100000
#define FF0     0
#define FF1     0100000
#define CBAUD   0000017
#define B0      0
#define B50     0000001
#define B75     0000002
#define B110    0000003
#define B134    0000004
#define B150    0000005
#define B200    0000006
#define B300    0000007
#define B600    0000010
```

(continued on next page)

Figure 6-42: `<termios.h>` (continued)

```
#define B1200   0000011
#define B1800   0000012
#define B2400   0000013
#define B4800   0000014
#define B9600   0000015
#define B19200  0000016
#define EXTA    0000016
#define B38400  0000017
#define EXTB    0000017
#define CSIZE   0000060
#define CS5     0
#define CS6     0000020
#define CS7     0000040
#define CS8     0000060
#define CSTOPB  0000100
#define CREAD   0000200
#define PARENB  0000400
#define PARODD  0001000
#define HUPCL   0002000
#define CLOCAL  0004000
#define CIBAUD  03600000
#define PAREXT  04000000
#define ISIG    0000001
#define ICANON  0000002
#define XCASE   0000004
#define ECHO    0000010
#define ECHOE   0000020
#define ECHOK   0000040
#define ECHONL  0000100
#define NOFLSH  0000200
#define TOSTOP  0000400
#define ECHOCTL 0001000
#define ECHOPRT 0002000
#define ECHOKE  0004000
#define FLUSHO  0020000
#define PENDIN  0040000
#define IEXTEN  0100000
#define TIOC    ('T'<<8)

#define TCSANOW (TIOC|14)
#define TCSADRAIN       (TIOC|15)
#define TCSAFLUSH       (TIOC|16)

#define TCIFLUSH 0
#define TCOFLUSH 1
#define TCIOFLUSH       2
#define TCOOFF  0
#define TCOON   1
#define TCIOFF  2
#define TCION   3

struct termios {
        tcflag_tc_iflag;
```

(continued on next page)

Figure 6-42: `<termios.h>` (continued)

```
        tcflag_tc_oflag;
        tcflag_tc_cflag;
        tcflag_tc_lflag;
        cc_t    c_cc[NCCS];
};

struct winsize {
        unsigned short  ws_row;
        unsigned short  ws_col;
        unsigned short  ws_xpixel;
        unsigned short  ws_ypixel;
};
```

Figure 6-43: `<sys/ticlts.h>`

```
#define TCL_BADADDR      1
#define TCL_BADOPT               2
#define TCL_NOPEER               3
#define TCL_PEERBADSTATE 4

#define TCL_DEFAULTADDRSZ        4
```

Figure 6-44: `<sys/ticots.h>`

```
#define TCO_NOPEER                       ECONNREFUSED
#define TCO_PEERNOROOMONQ                ECONNREFUSED
#define TCO_PEERBADSTATE         ECONNREFUSED
#define TCO_PEERINITIATED                ECONNRESET
#define TCO_PROVIDERINITIATED    ECONNABORTED

#define TCO_DEFAULTADDRSZ                4
```

Figure 6-45: `<sys/ticotsord.h>`

```
#define TCOO_NOPEER             1
#define TCOO_PEERNOROOMONQ              2
#define TCOO_PEERBADSTATE               3
#define TCOO_PEERINITIATED              4
#define TCOO_PROVIDERINITIATED  5

#define TCOO_DEFAULTADDRSZ              4
```

Figure 6-46: `<sys/tihdr.h>`

```
#define T_INFO_REQ              5
#define T_BIND_REQ              6
#define T_UNBIND_REQ    7
#define T_OPTMGMT_REQ   9

#define T_INFO_ACK              16
#define T_BIND_ACK              17
#define T_OK_ACK        19
#define T_OPTMGMT_ACK   22
```

Figure 6-47: `<sys/time.h>`

```
#define CLK_TCK          *
#define CLOCKS_PER_SEC   1000000
#define NULL             0

typedef long clock_t;
typedef long time_t;

struct tm {
        int tm_sec;
        int tm_min;
        int tm_hour;
        int tm_mday;
        int tm_mon;
        int tm_year;
        int tm_wday;
        int tm_yday;
        int tm_isdst;
};
struct timeval {
        time_t  tv_sec;
        long    tv_usec;
};

extern long timezone;
extern int daylight;
extern char *tzname[2];

typedef struct timestruc {
        time_t          tv_sec;
        long            tv_nsec;
} timestruc_t;

/* starred values may vary and should be
        retrieved with sysconf() of pathconf() */
```

Figure 6-48: `<sys/times.h>`

```
struct tms {
        clock_t tms_utime;
        clock_t tms_stime;
        clock_t tms_cutime;
        clock_t tms_cstime;
};
```

Figure 6-49: `<sys/timod.h>`

```
#define TIMOD           ('T'<<8)
#define TI_GETINFO      (TIMOD|140)
#define TI_OPTMGMT      (TIMOD|141)
#define TI_BIND         (TIMOD|142)
#define TI_UNBIND               (TIMOD|143)
#define TI_GETMYNAME    (TIMOD|144)
#define TI_GETPEERNAME  (TIMOD|145)
#define TI_SETMYNAME    (TIMOD|146)
#define TI_SETPEERNAME  (TIMOD|147)
```

Figure 6-50: `<sys/tiuser.h>`**, Service Types**

```
#define T_CLTS          3
#define T_COTS          1
#define T_COTS_ORD      2
```

Figure 6-51: `<tiuser.h>`**, Transport Interface States**

```
#define T_DATAXFER  5
#define T_IDLE      2
#define T_INCON     4
#define T_INREL     7
#define T_OUTCON    3
#define T_OUTREL    6
#define T_UNBND     1
#define T_UNINIT    0
```

Figure 6-52: `<sys/tiuser.h>`**, User-level Events**

```
#define T_ACCEPT1    12
#define T_ACCEPT2    13
#define T_ACCEPT3    14
#define T_BIND       1
#define T_CLOSE      4
#define T_CONNECT1   8
#define T_CONNECT2   9
#define T_LISTN      11
#define T_OPEN       0
#define T_OPTMGMT    2
#define T_PASSCON    24
#define T_RCV        16
#define T_RCVCONNECT 10
#define T_RCVDIS1    19
#define T_RCVDIS2    20
#define T_RCVDIS3    21
#define T_RCVREL     23
#define T_RCVUDATA   6
#define T_RCVUDERR   7
#define T_SND        15
#define T_SNDDIS1    17
#define T_SNDDIS2    18
#define T_SNDREL     22
#define T_SNDUDATA   5
#define T_UNBIND     3
```

Figure 6-53: `<sys/tiuser.h>`**, Error Return Values**

```
#define TACCES      3
#define TBADADDR    1
#define TBADDATA    10
#define TBADF       4
#define TBADFLAG    16
#define TBADOPT     2
#define TBADSEQ     7
#define TBUFOVFLW   11
#define TFLOW       12
#define TLOOK       9
#define TNOADDR     5
#define TNODATA     13
#define TNODIS      14
#define TNOREL      17
#define TNOTSUPPORT 18
#define TNOUDERR    15
#define TOUTSTATE   6
#define TSTATECHNG  19
#define TSYSERR     8
```

Figure 6-54: `<sys/tiuser.h>`**, Transport Interface Data Structures**

```
struct netbuf {
        unsigned int    maxlen;
        unsigned int    len;
        char            *buf;
};

struct t_bind {
        struct netbuf   addr;
        unsigned int    qlen;
};

struct t_call {
        struct netbuf   addr;
        struct netbuf   opt;
        struct netbuf   udata;
        int             sequence;
};

struct t_discon {
        struct netbuf   udata;
        int             reason;
        int             sequence;
};
```

(continued on next page)

Figure 6-54: `<sys/tiuser.h>`**, Transport Interface Data Structures** (continued)

```
struct t_info {
        long    addr;
        long    options;
        long    tsdu;
        long    etsdu;
        long    connect;
        long    discon;
        long    servtype;
};

struct t_optmgmt {
        struct netbuf   opt;
        long            flags;
};

struct t_uderr {
        struct netbuf   addr;
        struct netbuf   opt;
        long            error;
};

struct t_unitdata {
        struct netbuf   addr;
        struct netbuf   opt;
        struct netbuf   udata;
};
```

Figure 6-55: `<sys/tiuser.h>`**, Structure Types**

```
#define T_BIND      1
#define T_CALL      3
#define T_DIS       4
#define T_INFO      7
#define T_OPTMGMT   2
#define T_UDERROR   6
#define T_UNITDATA  5
```

Figure 6-56: `<sys/tiuser.h>`**, Fields of Structures**

```
#define T_ADDR      0x01
#define T_OPT       0x02
#define T_UDATA     0x04
#define T_ALL       0x07
```

Figure 6-57: `<sys/tiuser.h>`**, Events Bitmasks**

```
#define T_LISTEN        0x01
#define T_CONNECT       0x02
#define T_DATA          0x04
#define T_EXDATA        0x08
#define T_DISCONNECT    0x10
#define T_ERROR         0x20
#define T_UDERR         0x40
#define T_ORDREL        0x80
#define T_EVENTS        0xff
```

Figure 6-58: `<sys/tiuser.h>`**, Flags**

```
#define T_MORE          0x01
#define T_EXPEDITED     0x02
#define T_NEGOTIATE     0x04
#define T_CHECK         0x08
#define T_DEFAULT       0x10
#define T_SUCCESS       0x20
#define T_FAILURE       0x40
```

Figure 6-59: `<sys/types.h>`

```
typedef long            time_t;
typedef long            daddr_t;
typedef unsigned long   dev_t;
typedef long            gid_t;
typedef unsigned long   ino_t;
typedef int             key_t;
typedef long            pid_t;
typedef unsigned long   mode_t;
typedef unsigned long   nlink_t;
typedef long            off_t;
typedef long            uid_t;
typedef unsigned int    size_t;
typedef long            clock_t;
```

Figure 6-60: `<ucontext.h>`

```
typedef int gregset_t[19];

struct fpq {
        unsigned long *fpq_addr;
        unsigned long fpq_instr;
};

struct  fq {
        union {
                double  whole;
                struct fpq fpq;
        } FQu;
};

struct  fpu {
        union {
                unsigned        fpu_regs[32];
                double          fpu_dregs[16];
        } fpu_fr;
        struct fq               *fpu_q;
        unsigned        fpu_fsr;
        unsigned char   fpu_qcnt;
        unsigned char   fpu_q_entrysize;
        unsigned char   fpu_en;
};

typedef struct fpu      fpregset_t;

typedef struct {
        gregset_t               gregs;
```

(continued on next page)

Figure 6-60: <ucontext.h> (continued)

```
        gwindows_t      *gwins;
        fpregset_t      fpregs;
} mcontext_t;

typedef struct ucontext {
        unsigned long   uc_flags;
        struct ucontext *uc_link;
        sigset_t        uc_sigmask;
        stack_t         uc_stack;
        mcontext_t      uc_mcontext;
        long            uc_filler[44];
} ucontext_t;

#define SPARC_MAXREGWINDOW      31

struct gwindows {
        int             wbcnt;
        int             *spbuf[SPARC_MAXREGWINDOW];
        struct rwindow  wbuf[SPARC_MAXREGWINDOW];
};

struct rwindow {
        int             rw_local[8];
        int             rw_in[8];
};

typedef struct gwindows gwindows_t;
```

Figure 6-61: <sys/uio.h>

```
typedef struct iovec {
        char    *iov_base;
        int     iov_len;
} iovec_t;
```

Figure 6-62: `<ulimit.h>`

```
#define UL_GETFSIZE      1
#define UL_SETFSIZE      2
```

Figure 6-63: `<unistd.h>`

```
#define R_OK            4
#define W_OK            2
#define X_OK            1
#define F_OK            0

#define F_ULOCK         0
#define F_LOCK          1
#define F_TLOCK         2
#define F_TEST          3

#define SEEK_SET        0
#define SEEK_CUR        1
#define SEEK_END        2

#define _POSIX_JOB_CONTROL1
#define _POSIX_SAVED_IDS 1
#define _POSIX_VDISABLE  0

#define _POSIX_VERSION   *
#define _XOPEN_VERSION   *

/* starred values vary and should be retrieved using sysconf() or pathconf() */

#define _SC_ARG_MAX      1
#define _SC_CHILD_MAX    2
#define _SC_CLK_TCK      3
#define _SC_NGROUPS_MAX  4
#define _SC_OPEN_MAX     5
#define _SC_JOB_CONTROL  6
#define _SC_SAVED_IDS    7
#define _SC_VERSION      8
#define _SC_PASS_MAX     9
#define _SC_LOGNAME_MAX  10
#define _SC_PAGESIZE     11
#define _SC_XOPEN_VERSION12

#define _PC_LINK_MAX     1
#define _PC_MAX_CANON    2
#define _PC_MAX_INPUT    3
#define _PC_NAME_MAX     4
#define _PC_PATH_MAX     5
#define _PC_PIPE_BUF     6
#define _PC_NO_TRUNC     7
#define _PC_VDISABLE     8
#define _PC_CHOWN_RESTRICTED9

#define STDIN_FILENO     0
#define STDOUT_FILENO    1
#define STDERR_FILENO    2
```

Figure 6-64: `<utime.h>`

```
struct utimbuf {
        time_t  actime;
        time_t  modtime;
};
```

Figure 6-65: `<sys/utsname.h>`

```
#define SYS_NMLN        257

struct utsname {
        char    sysname[SYS_NMLN];
        char    nodename[SYS_NMLN];
        char    release[SYS_NMLN];
        char    version[SYS_NMLN];
        char    machine[SYS_NMLN];
};
```

Figure 6-66: `<wait.h>`

```
#define WEXITED         0001
#define WTRAPPED        0002
#define WSTOPPED        0004
#define WCONTINUED      0010
#define WUNTRACED       WSTOPPED
#define WNOHANG         0100
#define WNOWAIT         0200

#define WSTOPFLAG       0177
#define WCONTFLAG       0177777
#define WCOREFLAG       0200
#define WSIGMASK        0177

#define WLOBYTE(stat)   ((int)((stat)&0377))
#define WHIBYTE(stat)   ((int)(((stat)>>8)&0377))
#define WWORD(stat)     ((int)((stat))&0177777)

#define WIFEXITED(stat)  (WLOBYTE(stat)==0)
#define WIFSIGNALED(stat)(WLOBYTE(stat)>0&&WHIBYTE(stat)==0)
#define WIFSTOPPED(stat) (WLOBYTE(stat)==WSTOPFLG&&WHIBYTE(stat)!=0)
#define WIFCONTINUED(stat)(WWORD(stat)==WCONTFLG)
#define WEXITSTATUS(stat)WHIBYTE(stat)
#define WTERMSIG(stat)  (WLOBYTE(stat)&WSIGMASK)
#define WSTOPSIG(stat)  WHIBYTE(stat)
#define WCOREDUMP(stat)  ((stat)&WCOREFLG)
```

X Window Data Definitions

NOTE This section is new, but will not be diffmarked.

This section contains standard data definitions that describe system data for the optional X Window System libraries specified in the Generic ABI. These data definitions are referred to by their names in angle brackets: <*name.h*> and <sys/*name.h*>. Included in these data definitions are macro definitions and structure definitions. While an ABI-conforming system may provide X11 and X Toolkit Intrinsics interfaces, it need not contain the actual data definitions referenced here. Programmers should observe that the sources of the structures defined in these data definitions are defined in SVID or the appropriate X Consortium documentation (see chapter 10 in the Generic ABI).

Figure 6-67: <X11/Composite.h>

```
extern WidgetClass compositeWidgetClass;
```

Figure 6-68: <X11/Constraint.h>

```
extern WidgetClass constraintWidgetClass;
```

Figure 6-69: <X11/Core.h>

```
extern WidgetClass coreWidgetClass;
```

Figure 6-70: `<X11/cursorfont.h>`**, Part 1 of 3**

```
#define XC_num_glyphs            154
#define XC_X_cursor              0
#define XC_arrow                 2
#define XC_based_arrow_down              4
#define XC_based_arrow_up                6
#define XC_boat                  8
#define XC_bogosity              10
#define XC_bottom_left_corner    12
#define XC_bottom_right_corner   14
#define XC_bottom_side           16
#define XC_bottom_tee            18
#define XC_box_spiral            20
#define XC_center_ptr            22
#define XC_circle                        24
#define XC_clock                 26
#define XC_coffee_mug            28
#define XC_cross                 30
#define XC_cross_reverse         32
#define XC_crosshair             34
#define XC_diamond_cross         36
#define XC_dot                   38
#define XC_dotbox                        40
#define XC_double_arrow          42
#define XC_draft_large           44
#define XC_draft_small           46
#define XC_draped_box            48
#define XC_exchange              50
#define XC_fleur                 52
#define XC_gobbler                       54
#define XC_gumby                 56
#define XC_hand1                 58
#define XC_hand2                 60
```

Figure 6-71: `<X11/cursorfont.h>`**, Part 2 of 3**

```
#define XC_heart                62
#define XC_icon                 64
#define XC_iron_cross           66
#define XC_left_ptr             68
#define XC_left_side            70
#define XC_left_tee             72
#define XC_leftbutton           74
#define XC_ll_angle             76
#define XC_lr_angle             78
#define XC_man                  80
#define XC_middlebutton         82
#define XC_mouse                        84
#define XC_pencil                       86
#define XC_pirate                       88
#define XC_plus                 90
#define XC_question_arrow               92
#define XC_right_ptr            94
#define XC_right_side           96
#define XC_right_tee            98
#define XC_rightbutton          100
#define XC_rtl_logo             102
#define XC_sailboat             104
#define XC_sb_down_arrow        106
#define XC_sb_h_double_arrow            108
#define XC_sb_left_arrow        110
#define XC_sb_right_arrow               112
#define XC_sb_up_arrow          114
#define XC_sb_v_double_arrow            116
#define XC_shuttle                      118
#define XC_sizing                       120
#define XC_spider                       122
#define XC_spraycan             124
```

Figure 6-72: `<X11/cursorfont.h>`**, Part 3 of 3**

```
#define XC_star                 126
#define XC_target                       128
#define XC_tcross                       130
#define XC_top_left_arrow               132
#define XC_top_left_corner              134
#define XC_top_right_corner             136
#define XC_top_side             138
#define XC_top_tee                      140
#define XC_trek                 142
#define XC_ul_angle             144
#define XC_umbrella             146
#define XC_ur_angle             148
#define XC_watch                150
#define XC_xterm                152
```

Figure 6-73: `<X11/Intrinsic.h>`**, Part 1 of 6**

```
typedef char     *String;

#define XtNumber(arr)\
        ((Cardinal) (sizeof(arr) / sizeof(arr[0])))

typedef void            *Widget;
typedef Widget          *WidgetList;
typedef void            *CompositeWidget;
typedef void            *WidgetClass;
typedef XtActionsRec    *XtActionList;

typedef void            *XtAppContext;
typedef unsigned long   XtValueMask;
typedef unsigned long   XtIntervalId;
typedef unsigned long   XtInputId;
typedef unsigned long   XtWorkProcId;
typedef unsigned int    XtGeometryMask;
typedef unsigned long   XtGCMask;
typedef unsigned long   Pixel;
typedef int             XtCacheType;
#define XtCacheNone     0x001
#define XtCacheAll      0x002
#define XtCacheByDisplay 0x003
#define XtCacheRefCount 0x100

typedef char            Boolean;
typedef long            XtArgVal;
typedef unsigned char   XtEnum;

typedef unsigned int    Cardinal;
typedef unsigned short  Dimension;
typedef short           Position;

typedef void            *XtPointer;
```

Figure 6-74: `<X11/Intrinsic.h>`**, Part 2 of 6**

```
typedef void             *XtTranslations;
typedef void             *XtAccelerators;
typedef unsigned int     Modifiers;

#define XtCWQueryOnly    (1 << 7)
#define XtSMDontChange   5

typedef void             *XtCacheRef;
typedef void             *XtActionHookId;
typedef unsigned long    EventMask;
typedef enum {XtListHead, XtListTail } XtListPosition;
typedef unsigned long    XtInputMask;

typedef struct {
        String           string;
        XtActionProc     proc;
} XtActionsRec;

typedef enum {
        XtAddress,
        XtBaseOffset,
        XtImmediate,
        XtResourceString,
        XtResourceQuark,
        XtWidgetBaseOffset,
        XtProcedureArg
} XtAddressMode;

typedef struct {
        XtAddressMode    address_mode;
        XtPointer                address_id;
        Cardinal         size;
} XtConvertArgRec, *XtConvertArgList;
```

Figure 6-75: `<X11/Intrinsic.h>`**, Part 3 of 6**

```
#define XtInputNoneMask          0L
#define XtInputReadMask          (1L<<0)
#define XtInputWriteMask         (1L<<1)
#define XtInputExceptMask                (1L<<2)

typedef struct {
        XtGeometryMask  request_mode;
        Position        x, y;
        Dimension               width, height, border_width;
        Widget          sibling;
} XtWidgetGeometry;

typedef struct {
        String          name;
        XtArgVal        value;
} Arg, *ArgList;

typedef XtPointer       XtVarArgsList;

typedef struct {
        XtCallbackProc  callback;
        XtPointer               closure;
} XtCallbackRec, *XtCallbackList;

typedef enum {
        XtCallbackNoList,
        XtCallbackHasNone,
        XtCallbackHasSome
} XtCallbackStatus;

typedef struct {
        Widget          shell_widget;
        Widget          enable_widget;
} XtPopdownIDRec, *XtPopdownID;
```

Figure 6-76: `<X11/Intrinsic.h>`**, Part 4 of 6**

```
typedef enum {
        XtGeometryYes,
        XtGeometryNo,
        XtGeometryAlmost,
        XtGeometryDone
} XtGeometryResult;

typedef enum {
        XtGrabNone,
        XtGrabNonexclusive,
        XtGrabExclusive
} XtGrabKind;

typedef struct {
        String          resource_name;
        String          resource_class;
        String          resource_type;
        Cardinal        resource_size;
        Cardinal        resource_offset;
        String          default_type;
        XtPointer               default_addr;
} XtResource, *XtResourceList;

typedef struct {
        char            match;
        String          substitution;
} SubstitutionRec, *Substitution;

typedef Boolean         (*XtFilePredicate);
typedef XtPointer               XtRequestId;

extern XtConvertArgRec const colorConvertArgs[];
extern XtConvertArgRec const screenConvertArg[];
```

Figure 6-77: `<X11/Intrinsic.h>`**, Part 5 of 6**

```
#define XtAllEvents      ((EventMask) -1L)
#define XtIMXEvent                       1
#define XtIMTimer                        2
#define XtIMAlternateInput               4
#define XtIMAll (XtIMXEvent | XtIMTimer | XtIMAlternateInput)

#define XtOffsetOf(s_type,field) XtOffset(s_type*,field)
#define XtNew(type)      ((type *) XtMalloc((unsigned) sizeof(type)))
#define XT_CONVERT_FAIL          (Atom)0x80000001

#define XtIsRectObj(object) \
        (_XtCheckSubclassFlag(object,(XtEnum)0x02))
#define XtIsWidget(object) \
        (_XtCheckSubclassFlag(object,(XtEnum)0x04))
#define XtIsComposite(widget) \
        (_XtCheckSubclassFlag(widget,(XtEnum)0x08))
#define XtIsConstraint(widget) \
        (_XtCheckSubclassFlag(widget,(XtEnum)0x10))
#define XtIsShell(widget) \
        (_XtCheckSubclassFlag(widget,(XtEnum)0x20))
#define XtIsOverrideShell(widget) \
        (_XtIsSubclassOf(widget,(WidgetClass)overrideShellWidgetClass,\
        (WidgetClass)shellWidgetClass, (XtEnum)0x20))
#define XtIsWMShell(widget) \
        (_XtCheckSubclassFlag(widget,(XtEnum)0x40))
#define XtIsVendorShell(widget)  \
        (_XtIsSubclassOf(widget,(WidgetClass)vendorShellWidgetClass,\
        (WidgetClass)wmShellWidgetClass,(XtEnum)0x40))
#define XtIsTransientShell(widget) \
        (_XtIsSubclassOf(widget,(WidgetClass)transientShellWidgetClass,\
        (WidgetClass)wmShellWidgetClass, (XtEnum)0x40))
#define XtIsTopLevelShell(widget)\
        (_XtCheckSubclassFlag(widget, (XtEnum)0x80))
#define XtIsApplicationShell(widget)\
        (_XtIsSubclassOf(widget,(WidgetClass)applicationShellWidgetClass,\
        (WidgetClass)topLevelShellWidgetClass, (XtEnum)0x80))
```

Figure 6-78: `<X11/Intrinsic.h>`**, Part 6 of 6**

```
#define XtSetArg(arg,n,d)\
        ((void)( (arg).name = (n), (arg).value = (XtArgVal)(d) ))
#define XtOffset(p_type,field)\
        ((Cardinal) (((char *) (&(((p_type)NULL)->field))) - ((char *) NULL)))

#define XtVaNestedList           "XtVaNestedList"
#define XtVaTypedArg             "XtVaTypedArg"
#define XtUnspecifiedPixmap              ((Pixmap)2)
#define XtUnspecifiedShellInt    (-1)
#define XtUnspecifiedWindow              ((Window)2)
#define XtUnspecifiedWindowGroup  ((Window)3)
#define XtDefaultForeground              "XtDefaultForeground"
#define XtDefaultBackground              "XtDefaultBackground"
#define XtDefaultFont            "XtDefaultFont"
#define XtDefaultFontSet         "XtDefaultFontSet"
```

Figure 6-79: `<X11/Object.h>`

```
extern WidgetClass objectClass;
```

Figure 6-80: `<X11/RectObj.h>`

```
extern WidgetClass rectObjClass;
```

Figure 6-81: `<X11/Shell.h>`

```
extern WidgetClass shellWidgetClass;
extern WidgetClass overrideShellWidgetClass;
extern WidgetClass wmShellWidgetClass;
extern WidgetClass transientShellWidgetClass;
extern WidgetClass topLevelShellWidgetClass;
extern WidgetClass applicationShellWidgetClass;
```

Figure 6-82: <X11/Vendor.h>

```
extern WidgetClass vendorShellWidgetClass;
```

Figure 6-83: <X11/X.h>**, Part 1 of 12**

```
typedef unsigned long XID;

typedef XID Window;
typedef XID Drawable;
typedef XID Font;
typedef XID Pixmap;
typedef XID Cursor;
typedef XID Colormap;
typedef XID GContext;
typedef XID KeySym;

typedef unsigned long Atom;
typedef unsigned long VisualID;
typedef unsigned long Time;
typedef unsigned char KeyCode;

#define AllTemporary     0L
#define AnyButton                0L
#define AnyKey           0L
#define AnyPropertyType  0L
#define CopyFromParent   0L
#define CurrentTime      0L
#define InputFocus               1L
#define NoEventMask      0L
#define None             0L
#define NoSymbol         0L
#define ParentRelative   1L
#define PointerWindow    0L
#define PointerRoot      1L
```

Figure 6-84: `<X11/X.h>`**, Part 2 of 12**

```
#define KeyPressMask            (1L<<0)
#define KeyReleaseMask          (1L<<1)
#define ButtonPressMask         (1L<<2)
#define ButtonReleaseMask               (1L<<3)
#define EnterWindowMask         (1L<<4)
#define LeaveWindowMask         (1L<<5)
#define PointerMotionMask               (1L<<6)
#define PointerMotionHintMask   (1L<<7)
#define Button1MotionMask               (1L<<8)
#define Button2MotionMask               (1L<<9)
#define Button3MotionMask               (1L<<10)
#define Button4MotionMask               (1L<<11)
#define Button5MotionMask               (1L<<12)
#define ButtonMotionMask        (1L<<13)
#define KeymapStateMask         (1L<<14)
#define ExposureMask            (1L<<15)
#define VisibilityChangeMask            (1L<<16)
#define StructureNotifyMask             (1L<<17)
#define ResizeRedirectMask              (1L<<18)
#define SubstructureNotifyMask  (1L<<19)
#define SubstructureRedirectMask (1L<<20)
#define FocusChangeMask         (1L<<21)
#define PropertyChangeMask              (1L<<22)
#define ColormapChangeMask              (1L<<23)
#define OwnerGrabButtonMask             (1L<<24)
```

Figure 6-85: `<X11/X.h>`**, Part 3 of 12**

```
#define KeyPress                2
#define KeyRelease                      3
#define ButtonPress             4
#define ButtonRelease           5
#define MotionNotify            6
#define EnterNotify             7
#define LeaveNotify             8
#define FocusIn                 9
#define FocusOut                10
#define KeymapNotify            11
#define Expose                  12
#define GraphicsExpose          13
#define NoExpose                14
#define VisibilityNotify        15
#define CreateNotify            16
#define DestroyNotify           17
#define UnmapNotify             18
#define MapNotify                       19
#define MapRequest                      20
#define ReparentNotify          21
#define ConfigureNotify         22
#define ConfigureRequest        23
#define GravityNotify           24
#define ResizeRequest           25
#define CirculateNotify         26
#define CirculateRequest        27
#define PropertyNotify          28
#define SelectionClear          29
#define SelectionRequest        30
#define SelectionNotify         31
#define ColormapNotify          32
#define ClientMessage           33
#define MappingNotify           34
#define LASTEvent              35      /* must be bigger than any event # */
```

Figure 6-86: `<X11/X.h>`**, Part 4 of 12**

```
#define ShiftMask                        (1<<0)
#define LockMask                 (1<<1)
#define ControlMask              (1<<2)
#define Mod1Mask                 (1<<3)
#define Mod2Mask                 (1<<4)
#define Mod3Mask                 (1<<5)
#define Mod4Mask                 (1<<6)
#define Mod5Mask                 (1<<7)

#define Button1Mask              (1<<8)
#define Button2Mask              (1<<9)
#define Button3Mask              (1<<10)
#define Button4Mask              (1<<11)
#define Button5Mask              (1<<12)
#define AnyModifier              (1<<15)

#define Button1                  1
#define Button2                  2
#define Button3                  3
#define Button4                  4
#define Button5                  5

#define NotifyNormal             0
#define NotifyGrab                       1
#define NotifyUngrab             2
#define NotifyWhileGrabbed               3
#define NotifyHint                       1
#define NotifyAncestor           0
#define NotifyVirtual            1
#define NotifyInferior           2
#define NotifyNonlinear          3
#define NotifyNonlinearVirtual   4
#define NotifyPointer            5
#define NotifyPointerRoot                6
#define NotifyDetailNone         7
```

Figure 6-87: `<X11/X.h>`**, Part 5 of 12**

```
#define VisibilityUnobscured            0
#define VisibilityPartiallyObscured     1
#define VisibilityFullyObscured  2

#define PlaceOnTop                      0
#define PlaceOnBottom           1

#define PropertyNewValue        0
#define PropertyDelete          1

#define ColormapUninstalled             0
#define ColormapInstalled               1

#define GrabModeSync            0
#define GrabModeAsync           1

#define GrabSuccess             0
#define AlreadyGrabbed          1
#define GrabInvalidTime         2
#define GrabNotViewable         3
#define GrabFrozen                      4

#define AsyncPointer            0
#define SyncPointer             1
#define ReplayPointer           2
#define AsyncKeyboard           3
#define SyncKeyboard            4
#define ReplayKeyboard          5
#define AsyncBoth                       6
#define SyncBoth                7

#define RevertToNone            (int)None
#define RevertToPointerRoot             (int)PointerRoot
#define RevertToParent          2
```

Figure 6-88: `<X11/X.h>`**, Part 6 of 12**

```
#define Success         0
#define BadRequest              1
#define BadValue        2
#define BadWindow               3
#define BadPixmap               4
#define BadAtom         5
#define BadCursor               6
#define BadFont         7
#define BadMatch        8
#define BadDrawable     9
#define BadAccess               10
#define BadAlloc        11
#define BadColor        12
#define BadGC           13
#define BadIDChoice     14
#define BadName         15
#define BadLength               16
#define BadImplementation       17

#define InputOutput     1
#define InputOnly               2

#define CWBackPixmap    (1L<<0)
#define CWBackPixel     (1L<<1)
#define CWBorderPixmap  (1L<<2)
#define CWBorderPixel   (1L<<3)
#define CWBitGravity    (1L<<4)
#define CWWinGravity    (1L<<5)
#define CWBackingStore  (1L<<6)
#define CWBackingPlanes (1L<<7)
#define CWBackingPixel  (1L<<8)
#define CWOverrideRedirect      (1L<<9)
#define CWSaveUnder     (1L<<10)
#define CWEventMask     (1L<<11)
#define CWDontPropagate (1L<<12)
#define CWColormap              (1L<<13)
#define CWCursor        (1L<<14)
```

Figure 6-89: `<X11/X.h>`**, Part 7 of 12**

```
#define CWX                         (1<<0)
#define CWY                         (1<<1)
#define CWWidth                     (1<<2)
#define CWHeight                    (1<<3)
#define CWBorderWidth               (1<<4)
#define CWSibling                           (1<<5)
#define CWStackMode                 (1<<6)

#define ForgetGravity               0
#define NorthWestGravity            1
#define NorthGravity                2
#define NorthEastGravity            3
#define WestGravity                 4
#define CenterGravity               5
#define EastGravity                 6
#define SouthWestGravity            7
#define SouthGravity                8
#define SouthEastGravity            9
#define StaticGravity               10
#define UnmapGravity                0

#define NotUseful                           0
#define WhenMapped                          1
#define Always                      2

#define IsUnmapped                          0
#define IsUnviewable                1
#define IsViewable                          2

#define SetModeInsert               0
#define SetModeDelete               1

#define DestroyAll                          0
#define RetainPermanent             1
#define RetainTemporary             2
```

Figure 6-90: `<X11/X.h>`**, Part 8 of 12**

```
#define Above                    0
#define Below                    1
#define TopIf                    2
#define BottomIf                 3
#define Opposite                 4
#define RaiseLowest              0
#define LowerHighest             1
#define PropModeReplace          0
#define PropModePrepend          1
#define PropModeAppend           2

#define GXclear                  0x0
#define GXand                    0x1
#define GXandReverse             0x2
#define GXcopy                   0x3
#define GXandInverted            0x4
#define GXnoop                   0x5
#define GXxor                    0x6
#define GXor                     0x7
#define GXnor                    0x8
#define GXequiv                  0x9
#define GXinvert                 0xa
#define GXorReverse              0xb
#define GXcopyInverted           0xc
#define GXorInverted             0xd
#define GXnand                   0xe
#define GXset                    0xf

#define LineSolid                        0
#define LineOnOffDash            1
#define LineDoubleDash           2
#define CapNotLast                       0
#define CapButt                  1
#define CapRound                 2
#define CapProjecting            3
```

Figure 6-91: `<X11/X.h>`**, Part 9 of 12**

```
#define JoinMiter                0
#define JoinRound                1
#define JoinBevel                2

#define FillSolid                0
#define FillTiled                1
#define FillStippled     2
#define FillOpaqueStippled       3

#define EvenOddRule      0
#define WindingRule      1

#define ClipByChildren   0
#define IncludeInferiors 1

#define Unsorted         0
#define YSorted          1
#define YXSorted         2
#define YXBanded         3

#define CoordModeOrigin  0
#define CoordModePrevious        1

#define Complex          0
#define Nonconvex                1
#define Convex           2

#define ArcChord         0
#define ArcPieSlice      1
```

Figure 6-92: `<X11/X.h>`**, Part 10 of 12**

```
#define GCFunction                      (1L<<0)
#define GCPlaneMask             (1L<<1)
#define GCForeground            (1L<<2)
#define GCBackground            (1L<<3)
#define GCLineWidth             (1L<<4)
#define GCLineStyle             (1L<<5)
#define GCCapStyle                      (1L<<6)
#define GCJoinStyle             (1L<<7)
#define GCFillStyle             (1L<<8)
#define GCFillRule                      (1L<<9)
#define GCTile                  (1L<<10)
#define GCStipple                       (1L<<11)
#define GCTileStipXOrigin               (1L<<12)
#define GCTileStipYOrigin               (1L<<13)
#define GCFont                  (1L<<14)
#define GCSubwindowMode         (1L<<15)
#define GCGraphicsExposures             (1L<<16)
#define GCClipXOrigin           (1L<<17)
#define GCClipYOrigin           (1L<<18)
#define GCClipMask                      (1L<<19)
#define GCDashOffset            (1L<<20)
#define GCDashList                      (1L<<21)
#define GCArcMode                       (1L<<22)

#define FontLeftToRight         0
#define FontRightToLeft         1

#define XYBitmap                0
#define XYPixmap                1
#define ZPixmap                 2

#define AllocNone                       0
#define AllocAll                1

#define DoRed                   (1<<0)
#define DoGreen                 (1<<1)
#define DoBlue                  (1<<2)
```

Figure 6-93: `<X11/X.h>`**, Part 11 of 12**

```
#define CursorShape             0
#define TileShape                       1
#define StippleShape            2

#define AutoRepeatModeOff               0
#define AutoRepeatModeOn        1
#define AutoRepeatModeDefault   2

#define LedModeOff                      0
#define LedModeOn                       1

#define KBKeyClickPercent               (1L<<0)
#define KBBellPercent           (1L<<1)
#define KBBellPitch             (1L<<2)
#define KBBellDuration          (1L<<3)
#define KBLed                   (1L<<4)
#define KBLedMode                       (1L<<5)
#define KBKey                   (1L<<6)
#define KBAutoRepeatMode        (1L<<7)

#define MappingSuccess          0
#define MappingBusy             1
#define MappingFailed           2

#define MappingModifier         0
#define MappingKeyboard         1
#define MappingPointer          2
#define DontPreferBlanking              0
#define PreferBlanking          1
#define DefaultBlanking         2

#define DontAllowExposures              0
#define AllowExposures          1
#define DefaultExposures        2
```

Figure 6-94: `<X11/X.h>`**, Part 12 of 12**

```
#define ScreenSaverReset 0
#define ScreenSaverActive        1

#define EnableAccess     1
#define DisableAccess    0
#define StaticGray               0
#define GrayScale                1

#define StaticColor      2
#define PseudoColor      3
#define TrueColor                4
#define DirectColor      5

#define LSBFirst         0
#define MSBFirst         1
```

Figure 6-95: `<X11/Xatom.h>`**, Part 1 of 3**

```
#define XA_PRIMARY                      ((Atom) 1)
#define XA_SECONDARY            ((Atom) 2)
#define XA_ARC                  ((Atom) 3)
#define XA_ATOM                 ((Atom) 4)
#define XA_BITMAP                       ((Atom) 5)
#define XA_CARDINAL             ((Atom) 6)
#define XA_COLORMAP             ((Atom) 7)
#define XA_CURSOR                       ((Atom) 8)
#define XA_CUT_BUFFER0          ((Atom) 9)
#define XA_CUT_BUFFER1          ((Atom) 10)
#define XA_CUT_BUFFER2          ((Atom) 11)
#define XA_CUT_BUFFER3          ((Atom) 12)
#define XA_CUT_BUFFER4          ((Atom) 13)
#define XA_CUT_BUFFER5          ((Atom) 14)
#define XA_CUT_BUFFER6          ((Atom) 15)
#define XA_CUT_BUFFER7          ((Atom) 16)
#define XA_DRAWABLE             ((Atom) 17)
#define XA_FONT                 ((Atom) 18)
#define XA_INTEGER                      ((Atom) 19)
#define XA_PIXMAP                       ((Atom) 20)
#define XA_POINT                ((Atom) 21)
#define XA_RECTANGLE            ((Atom) 22)
#define XA_RESOURCE_MANAGER             ((Atom) 23)
#define XA_RGB_COLOR_MAP        ((Atom) 24)
#define XA_RGB_BEST_MAP         ((Atom) 25)
#define XA_RGB_BLUE_MAP         ((Atom) 26)
#define XA_RGB_DEFAULT_MAP              ((Atom) 27)
#define XA_RGB_GRAY_MAP         ((Atom) 28)
#define XA_RGB_GREEN_MAP        ((Atom) 29)
#define XA_RGB_RED_MAP          ((Atom) 30)
#define XA_STRING                       ((Atom) 31)
#define XA_VISUALID             ((Atom) 32)
```

Figure 6-96: `<X11/Xatom.h>`**, Part 2 of 3**

```
#define XA_WINDOW                        ((Atom) 33)
#define XA_WM_COMMAND            ((Atom) 34)
#define XA_WM_HINTS              ((Atom) 35)
#define XA_WM_CLIENT_MACHINE             ((Atom) 36)
#define XA_WM_ICON_NAME          ((Atom) 37)
#define XA_WM_ICON_SIZE          ((Atom) 38)
#define XA_WM_NAME                       ((Atom) 39)
#define XA_WM_NORMAL_HINTS               ((Atom) 40)
#define XA_WM_SIZE_HINTS         ((Atom) 41)
#define XA_WM_ZOOM_HINTS         ((Atom) 42)
#define XA_MIN_SPACE             ((Atom) 43)
#define XA_NORM_SPACE            ((Atom) 44)
#define XA_MAX_SPACE             ((Atom) 45)
#define XA_END_SPACE             ((Atom) 46)
#define XA_SUPERSCRIPT_X         ((Atom) 47)
#define XA_SUPERSCRIPT_Y         ((Atom) 48)
#define XA_SUBSCRIPT_X           ((Atom) 49)
#define XA_SUBSCRIPT_Y           ((Atom) 50)
#define XA_UNDERLINE_POSITION    ((Atom) 51)
#define XA_UNDERLINE_THICKNESS   ((Atom) 52)
#define XA_STRIKEOUT_ASCENT              ((Atom) 53)
#define XA_STRIKEOUT_DESCENT             ((Atom) 54)
#define XA_ITALIC_ANGLE          ((Atom) 55)
#define XA_X_HEIGHT              ((Atom) 56)
#define XA_QUAD_WIDTH            ((Atom) 57)
#define XA_WEIGHT                        ((Atom) 58)
#define XA_POINT_SIZE            ((Atom) 59)
#define XA_RESOLUTION            ((Atom) 60)
#define XA_COPYRIGHT             ((Atom) 61)
#define XA_NOTICE                        ((Atom) 62)
#define XA_FONT_NAME             ((Atom) 63)
#define XA_FAMILY_NAME           ((Atom) 64)
```

Figure 6-97: `<X11/Xatom.h>`**, Part 3 of 3**

```
#define XA_FULL_NAME            ((Atom) 65)
#define XA_CAP_HEIGHT           ((Atom) 66)
#define XA_WM_CLASS             ((Atom) 67)
#define XA_WM_TRANSIENT_FOR             ((Atom) 68)
#define XA_LAST_PREDEFINED              ((Atom) 68)
```

Figure 6-98: `<X11/Xcms.h>`**, Part 1 of 5**

```
#define XcmsFailure                     0
#define XcmsSuccess                     1
#define XcmsSuccessWithCompression              2

#define XcmsUndefinedFormat      (XcmsColorFormat)0x00000000
#define XcmsCIEXYZFormat         (XcmsColorFormat)0x00000001
#define XcmsCIEuvYFormat         (XcmsColorFormat)0x00000002
#define XcmsCIExyYFormat         (XcmsColorFormat)0x00000003
#define XcmsCIELabFormat         (XcmsColorFormat)0x00000004
#define XcmsCIELuvFormat         (XcmsColorFormat)0x00000005
#define XcmsTekHVCFormat         (XcmsColorFormat)0x00000006
#define XcmsRGBFormat            (XcmsColorFormat)0x80000000
#define XcmsRGBiFormat           (XcmsColorFormat)0x80000001

#define XcmsInitNone             0x00
#define XcmsInitSuccess          0x01

typedef unsigned int XcmsColorFormat;

typedef double XcmsFloat;

typedef struct {
        unsigned short red;
        unsigned short green;
        unsigned short blue;
} XcmsRGB;
```

Figure 6-99: `<X11/Xcms.h>`**, Part 2 of 5**

```
typedef struct {
        XcmsFloat red;
        XcmsFloat green;
        XcmsFloat blue;
} XcmsRGBi;

typedef struct {
        XcmsFloat X;
        XcmsFloat Y;
        XcmsFloat Z;
} XcmsCIEXYZ;

typedef struct {
        XcmsFloat u_prime;
        XcmsFloat v_prime;
        XcmsFloat Y;
} XcmsCIEuvY;

typedef struct {
        XcmsFloat x;
        XcmsFloat y;
        XcmsFloat Y;
} XcmsCIExyY;

typedef struct {
        XcmsFloat L_star;
        XcmsFloat a_star;
        XcmsFloat b_star;
} XcmsCIELab;
```

Figure 6-100: `<X11/Xcms.h>`**, Part 3 of 5**

```
typedef struct {
        XcmsFloat L_star;
        XcmsFloat u_star;
        XcmsFloat v_star;
} XcmsCIELuv;

typedef struct {
        XcmsFloat H;
        XcmsFloat V;
        XcmsFloat C;
} XcmsTekHVC;

typedef struct {
        XcmsFloat pad0;
        XcmsFloat pad1;
        XcmsFloat pad2;
        XcmsFloat pad3;
} XcmsPad;
```

Figure 6-101: `<X11/Xcms.h>`**, Part 4 of 5**

```
typedef struct {
        union {
                XcmsRGB         RGB;
                XcmsRGBi        RGBi;
                XcmsCIEXYZ      CIEXYZ;
                XcmsCIEuvY      CIEuvY;
                XcmsCIExyY      CIExyY;
                XcmsCIELab      CIELab;
                XcmsCIELuv      CIELuv;
                XcmsTekHVC      TekHVC;
                XcmsPad         Pad;
        }       spec;
        unsigned long   pixel;
        XcmsColorFormat format;
} XcmsColor;

typedef struct {
        XcmsColor               screenWhitePt;
        XPointer        functionSet;
        XPointer        screenData;
        unsigned char   state;
        char            pad[3];
} XcmsPerScrnInfo;

typedef void *XcmsCCC;

typedef Status (*XcmsConversionProc)();
typedef XcmsConversionProc *XcmsFuncListPtr;
```

Figure 6-102: `<X11/Xcms.h>`**, Part 5 of 5**

```
typedef struct {
        char                    *prefix;
        XcmsColorFormat         id;
        XcmsParseStringProc     parseString;
        XcmsFuncListPtr         to_CIEXYZ;
        XcmsFuncListPtr         from_CIEXYZ;
        int                     inverse_flag;
} XcmsColorSpace;

typedef struct {
        XcmsColorSpace          **DDColorSpaces;
        XcmsScreenInitProc              screenInitProc;
        XcmsScreenFreeProc              screenFreeProc;
} XcmsFunctionSet;
```

Figure 6-103: `<X11/Xlib.h>` **Part 1 of 27**

```
typedef void *XPointer;

#define Bool                    int
#define Status                  int
#define True                    1
#define False                   0
#define QueuedAlready           0
#define QueuedAfterReading              1
#define QueuedAfterFlush        2

#define AllPlanes                       ((unsigned long)~0L)
```

Figure 6-104: `<X11/Xlib.h>` **Part 2 of 27**

```
typdef void XExtData;

typdef void XExtCodes;

typedef struct {
        int depth;
        int bits_per_pixel;
        int scanline_pad;
} XPixmapFormatValues;
```

Figure 6-105: <X11/Xlib.h> **Part 3 of 27**

```
typedef struct {
        int function;
        unsigned long plane_mask;
        unsigned long foreground;
        unsigned long background;
        int line_width;
        int line_style;
        int cap_style;
        int join_style;
        int fill_style;
        int fill_rule;
        int arc_mode;
        Pixmap tile;
        Pixmap stipple;
        int ts_x_origin;
        int ts_y_origin;
        Font font;
        int subwindow_mode;
        Bool graphics_exposures;
        int clip_x_origin;
        int clip_y_origin;
        Pixmap clip_mask;
        int dash_offset;
        char dashes;
} XGCValues;

typedef void *GC;

typedef struct _dummy Visual;
```

Figure 6-106: <X11/Xlib.h> **Part 4 of 27**

```
typedef struct _dummy Screen;

typedef struct {
        Pixmap background_pixmap;
        unsigned long background_pixel;
        Pixmap border_pixmap;
        unsigned long border_pixel;
        int bit_gravity;
        int win_gravity;
        int backing_store;
        unsigned long backing_planes;
        unsigned long backing_pixel;
        Bool save_under;
        long event_mask;
        long do_not_propagate_mask;
        Bool override_redirect;
        Colormap colormap;
        Cursor cursor;
} XSetWindowAttributes;
```

Figure 6-107: `<X11/Xlib.h>` **Part 5 of 27**

```
typedef struct {
        XExtData *ext_data;
        int depth;
        int bits_per_pixel;
        int scanline_pad;
} ScreenFormat;

typedef struct {
        int x, y;
        int width, height;
        int border_width;
        int depth;
        Visual *visual;
        Window root;
        int class;
        int bit_gravity;
        int win_gravity;
        int backing_store;
        unsigned long backing_planes;
        unsigned long backing_pixel;
        Bool save_under;
        Colormap colormap;
        Bool map_installed;
        int map_state;
        long all_event_masks;
        long your_event_mask;
        long do_not_propagate_mask;
        Bool override_redirect;
        Screen *screen;
} XWindowAttributes;
```

Figure 6-108: `<X11/Xlib.h>` **Part 6 of 27**

```
typedef struct {
        int family;
        int length;
        char *address;
} XHostAddress;

typedef struct _XImage {
        int width, height;
        int xoffset;
        int format;
        char *data;
        int byte_order;
        int bitmap_unit;
        int bitmap_bit_order;
        int bitmap_pad;
        int depth;
        int bytes_per_line;
        int bits_per_pixel;
        unsigned long red_mask;
        unsigned long green_mask;
        unsigned long blue_mask;
        XPointer obdata;
        struct funcs {
                struct _XImage *(*create_image)();
                int (*destroy_image)();
                unsigned long (*get_pixel)();
                int (*put_pixel)();
                struct _XImage *(*sub_image)();
                int (*add_pixel)();
        } f;
} XImage;
```

Figure 6-109: `<X11/Xlib.h>` **Part 7 of 27**

```
typedef struct {
        int x, y;
        int width, height;
        int border_width;
        Window sibling;
        int stack_mode;
} XWindowChanges;

typedef struct {
        unsigned long pixel;
        unsigned short red, green, blue;
        char flags;
        char pad;
} XColor;

typedef struct {
        short x1, y1, x2, y2;
} XSegment;

typedef struct {
        short x, y;
} XPoint;

typedef struct {
        short x, y;
        unsigned short width, height;
} XRectangle;

typedef struct {
        short x, y;
        unsigned short width, height;
        short angle1, angle2;
} XArc;
```

Figure 6-110: <X11/Xlib.h> **Part 8 of 27**

```
typedef struct {
        int key_click_percent;
        int bell_percent;
        int bell_pitch;
        int bell_duration;
        int led;
        int led_mode;
        int key;
        int auto_repeat_mode;
} XKeyboardControl;

typedef struct {
        int key_click_percent;
        int bell_percent;
        unsigned int bell_pitch, bell_duration;
        unsigned long led_mask;
        int global_auto_repeat;
        char auto_repeats[32];
} XKeyboardState;
typedef struct {
        Time time;
        short x, y;
} XTimeCoord;

typedef struct {
        int             max_keypermod;
        KeyCode *modifiermap;
} XModifierKeymap;

typedef struct _dummy Display;
```

Figure 6-111: `<X11/Xlib.h>` **Part 9 of 27**

```
typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Window window;
        Window root;
        Window subwindow;
        Time time;
        int x, y;
        int x_root, y_root;
        unsigned int state;
        unsigned int keycode;
        Bool same_screen;
} XKeyEvent;
typedef XKeyEvent XKeyPressedEvent;
typedef XKeyEvent XKeyReleasedEvent;

typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Window window;
        Window root;
        Window subwindow;
        Time time;
        int x, y;
        int x_root, y_root;
        unsigned int state;
        unsigned int button;
        Bool same_screen;
} XButtonEvent;
typedef XButtonEvent XButtonPressedEvent;
typedef XButtonEvent XButtonReleasedEvent;
```

Figure 6-112: <X11/Xlib.h> **Part 10 of 27**

```
typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Window window;
        Window root;
        Window subwindow;
        Time time;
        int x, y;
        int x_root, y_root;
        unsigned int state;
        char is_hint;
        Bool same_screen;
} XMotionEvent;
typedef XMotionEvent XPointerMovedEvent;

typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Window window;
        Window root;
        Window subwindow;
        Time time;
        int x, y;
        int x_root, y_root;
        int mode;
        int detail;
        Bool same_screen;
        Bool focus;
        unsigned int state;
} XCrossingEvent;
```

Figure 6-113: `<X11/Xlib.h>` **Part 11 of 27**

```
typedef XCrossingEvent XEnterWindowEvent;
typedef XCrossingEvent XLeaveWindowEvent;
typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Window window;
        int mode;
        int detail;
} XFocusChangeEvent;
typedef XFocusChangeEvent XFocusInEvent;
typedef XFocusChangeEvent XFocusOutEvent;

typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Window window;
        char key_vector[32];
} XKeymapEvent;

typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Window window;
        int x, y;
        int width, height;
        int count;
} XExposeEvent;
```

Figure 6-114: `<X11/Xlib.h>` **Part 12 of 27**

```
typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Drawable drawable;
        int x, y;
        int width, height;
        int count;
        int major_code;
        int minor_code;
} XGraphicsExposeEvent;

typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Drawable drawable;
        int major_code;
        int minor_code;
} XNoExposeEvent;

typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Window window;
        int state;
} XVisibilityEvent;
```

Figure 6-115: `<X11/Xlib.h>` **Part 13 of 27**

```
typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Window parent;
        Window window;
        int x, y;
        int width, height;
        int border_width;
        Bool override_redirect;
} XCreateWindowEvent;

typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Window event;
        Window window;
} XDestroyWindowEvent;

typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Window event;
        Window window;
        Bool from_configure;
} XUnmapEvent;
```

Figure 6-116: <X11/Xlib.h> **Part 14 of 27**

```
typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Window event;
        Window window;
        Bool override_redirect;
} XMapEvent;

typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Window parent;
        Window window;
} XMapRequestEvent;

typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Window event;
        Window window;
        Window parent;
        int x, y;
        Bool override_redirect;
} XReparentEvent;
```

Figure 6-117: `<X11/Xlib.h>` **Part 15 of 27**

```
typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Window event;
        Window window;
        int x, y;
        int width, height;
        int border_width;
        Window above;
        Bool override_redirect;
} XConfigureEvent;

typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Window event;
        Window window;
        int x, y;
} XGravityEvent;

typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Window window;
        int width, height;
} XResizeRequestEvent;
```

Figure 6-118: <X11/Xlib.h> **Part 16 of 27**

```
typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Window parent;
        Window window;
        int x, y;
        int width, height;
        int border_width;
        Window above;
        int detail;
        unsigned long value_mask;
} XConfigureRequestEvent;

typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Window event;
        Window window;
        int place;
} XCirculateEvent;

typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Window parent;
        Window window;
        int place;
} XCirculateRequestEvent;
```

Figure 6-119: `<X11/Xlib.h>` **Part 17 of 27**

```
typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Window window;
        Atom atom;
        Time time;
        int state;
} XPropertyEvent;

typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Window window;
        Atom selection;
        Time time;
} XSelectionClearEvent;

typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Window owner;
        Window requestor;
        Atom selection;
        Atom target;
        Atom property;
        Time time;
} XSelectionRequestEvent;
```

Figure 6-120: <X11/Xlib.h> **Part 18 of 27**

```
typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Window requestor;
        Atom selection;
        Atom target;
        Atom property;
        Time time;
} XSelectionEvent;

typedef struct {
        int type;
        Display *display;
        XID resourceid;
        unsigned long serial;
        unsigned char error_code;
        unsigned char request_code;
        unsigned char minor_code;
} XErrorEvent;

typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Window window;
        Atom message_type;
        int format;
        union {
                char b[20];
                short s[10];
                long l[5];
        } data;
} XClientMessageEvent;
```

Figure 6-121: `<X11/Xlib.h>` **Part 19 of 27**

```
typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Window window;
        Colormap colormap;
        Bool new;
        int state;
} XColormapEvent;

typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Window window;
        int request;
        int first_keycode;
        int count;
} XMappingEvent;

typedef struct {
        int type;
        unsigned long serial;
        Bool send_event;
        Display *display;
        Window window;
} XAnyEvent;
```

Figure 6-122: <X11/Xlib.h> **Part 20 of 27**

```
typedef union _XEvent {
        int                     type;
        XAnyEvent                       xany;
        XKeyEvent                       xkey;
        XButtonEvent            xbutton;
        XMotionEvent            xmotion;
        XCrossingEvent          xcrossing;
        XFocusChangeEvent               xfocus;
        XExposeEvent            xexpose;
        XGraphicsExposeEvent    xgraphicsexpose;
        XNoExposeEvent          xnoexpose;
        XVisibilityEvent        xvisibility;
        XCreateWindowEvent              xcreatewindow;
        XDestroyWindowEvent     xdestroywindow;
        XUnmapEvent             xunmap;
        XMapEvent                       xmap;
        XMapRequestEvent        xmaprequest;
        XReparentEvent          xreparent;
        XConfigureEvent         xconfigure;
        XGravityEvent           xgravity;
        XResizeRequestEvent     xresizerequest;
        XConfigureRequestEvent  xconfigurerequest;
        XCirculateEvent         xcirculate;
        XCirculateRequestEvent  xcirculaterequest;
        XPropertyEvent          xproperty;
        XSelectionClearEvent    xselectionclear;
        XSelectionRequestEvent  xselectionrequest;
        XSelectionEvent         xselection;
        XColormapEvent          xcolormap;
        XClientMessageEvent     xclient;
        XMappingEvent           xmapping;
        XErrorEvent             xerror;
        XKeymapEvent            xkeymap;
        long                    pad[24];
} XEvent;
```

Figure 6-123: `<X11/Xlib.h>` **Part 21 of 27**

```
#define XAllocID(dpy) ((*(dpy)->resource_alloc)((dpy)))

typedef struct {
        short   lbearing;
        short   rbearing;
        short   width;
        short   ascent;
        short   descent;
        unsigned short attributes;
} XCharStruct;

typedef struct {
        Atom name;
        unsigned long card32;
} XFontProp;

typedef struct {
        XExtData*ext_data;
        Font            fid;
        unsigneddirection;
        unsignedmin_char_or_byte2;
        unsignedmax_char_or_byte2;
        unsignedmin_byte1;
        unsignedmax_byte1;
        Bool            all_chars_exist;
        unsigneddefault_char;
        int             n_properties;
        XFontProp       *properties;
        XCharStruct     min_bounds;
        XCharStruct     max_bounds;
        XCharStruct     *per_char;
        int             ascent;
        int             descent;
} XFontStruct;
```

Figure 6-124: <X11/Xlib.h> **Part 22 of 27**

```
typedef struct {
        char *chars;
        int nchars;
        int delta;
        Font font;
} XTextItem;

typedef struct {
        unsigned char byte1;
        unsigned char byte2;
} XChar2b;

typedef struct {
        XChar2b *chars;
        int nchars;
        int delta;
        Font font;
} XTextItem16;

typedef union {
        Display *display;
        GC gc;
        Visual *visual;
        Screen *screen;
        ScreenFormat *pixmap_format;
        XFontStruct *font;
} XEDataObject;

typedef struct {
        XRectangle      max_ink_extent;
        XRectangle      max_logical_extent;
} XFontSetExtents;

typedef struct _dummy XFontSet;
```

Figure 6-125: `<X11/Xlib.h>` **Part 23 of 27**

```
typedef struct {
        char            *chars;
        int             nchars;
        int             delta;
        XFontSet *font_set;
} XmbTextItem;

typedef struct {
        wchar_t *chars;
        int             nchars;
        int             delta;
        XFontSet font_set;
} XwcTextItem;

typedef void (*XIMProc)();

typedef void    *XIM;
typedef void    *XIC;

typedef unsigned long XIMStyle;

typedef struct {
        unsigned short count_styles;
        XIMStyle *supported_styles;
} XIMStyles;

#define XIMPreeditArea          0x0001L
#define XIMPreeditCallbacks     0x0002L
#define XIMPreeditPosition              0x0004L
#define XIMPreeditNothing               0x0008L
#define XIMPreeditNone          0x0010L
#define XIMStatusArea           0x0100L
#define XIMStatusCallbacks              0x0200L
#define XIMStatusNothing        0x0400L
#define XIMStatusNone           0x0800L
```

Figure 6-126: `<X11/Xlib.h>` **Part 24 of 27**

```
#define XNVaNestedList            "XNVaNestedList"
#define XNQueryInputStyle                 "queryInputStyle"
#define XNClientWindow            "clientWindow"
#define XNInputStyle              "inputStyle"
#define XNFocusWindow             "focusWindow"
#define XNResourceName            "resourceName"
#define XNResourceClass           "resourceClass"
#define XNGeometryCallback                "geometryCallback"
#define XNFilterEvents            "filterEvents"
#define XNPreeditStartCallback            "preeditStartCallback"
#define XNPreeditDoneCallback     "preeditDoneCallback"
#define XNPreeditDrawCallback     "preeditDrawCallback"
#define XNPreeditCaretCallback    "preeditCaretCallback"
#define XNPreeditAttributes               "preeditAttributes"
#define XNStatusStartCallback     "statusStartCallback"
#define XNStatusDoneCallback              "statusDoneCallback"
#define XNStatusDrawCallback      "statusDrawCallback"
#define XNStatusAttributes                        "statusAttributes"
#define XNArea                    "area"
#define XNAreaNeeded              "areaNeeded"
#define XNSpotLocation            "spotLocation"
#define XNColormap                        "colorMap"
#define XNStdColormap             "stdColorMap"
#define XNForeground              "foreground"
#define XNBackground              "background"
#define XNBackgroundPixmap                "backgroundPixmap"
#define XNFontSet                         "fontSet"
#define XNLineSpace               "lineSpace"
#define XNCursor                  "cursor"
```

Figure 6-127: `<X11/Xlib.h>` **Part 25 of 27**

```
#define XBufferOverflow  -1
#define XLookupNone      1
#define XLookupChars     2
#define XLookupKeySym    3
#define XLookupBoth      4

typedef XPointer XVaNestedList;

typedef struct {
        XPointer client_data;
        XIMProc callback;
} XIMCallback;

typedef unsigned long XIMFeedback;

#define XIMReverse               1
#define XIMUnderline     (1<<1)
#define XIMHighlight     (1<<2)
#define XIMPrimary               (1<<5)
#define XIMSecondary     (1<<6)
#define XIMTertiary      (1<<7)

typedef struct _XIMText {
        unsigned short length;
        XIMFeedback *feedback;
        Bool encoding_is_wchar;
        union {
                char *multi_byte;
                wchar_t *wide_char;
        } string;
} XIMText;
```

Figure 6-128: <X11/Xlib.h> **Part 26 of 27**

```
typedef struct _XIMPreeditDrawCallbackStruct {
        int caret;
        int chg_first;
        int chg_length;
        XIMText *text;
} XIMPreeditDrawCallbackStruct;

typedef enum {
        XIMForwardChar, XIMBackwardChar,
        XIMForwardWord, XIMBackwardWord,
        XIMCaretUp, XIMCaretDown,
        XIMNextLine, XIMPreviousLine,
        XIMLineStart, XIMLineEnd,
        XIMAbsolutePosition,
        XIMDontChange
} XIMCaretDirection;

typedef enum {
        XIMIsInvisible,
        XIMIsPrimary,
        XIMIsSecondary
} XIMCaretStyle;

typedef struct _XIMPreeditCaretCallbackStruct {
        int position;
        XIMCaretDirection direction;
        XIMCaretStyle style;
} XIMPreeditCaretCallbackStruct;
```

Figure 6-129: <X11/Xlib.h> **Part 27 of 27**

```
typedef enum {
        XIMTextType,
        XIMBitmapType
} XIMStatusDataType;

typedef struct _XIMStatusDrawCallbackStruct {
        XIMStatusDataType type;
        union {
                XIMText *text;
                Pixmap  bitmap;
        } data;
} XIMStatusDrawCallbackStruct;
```

Figure 6-130: `<X11/Xresource.h>`**, Part 1 of 2**

```
typedef int             XrmQuark, *XrmQuarkList;
#define NULLQUARK       ((XrmQuark) 0)

typedef enum {XrmBindTightly, XrmBindLoosely} \
                XrmBinding, *XrmBindingList;

typedef XrmQuark                        XrmName;
typedef XrmQuarkList                    XrmNameList;
typedef XrmQuark                        XrmClass;
typedef XrmQuarkList                    XrmClassList;
typedef XrmQuark                        XrmRepresentation;

#define XrmStringToName(string)         XrmStringToQuark(string)
#define XrmStringToNameList(str, name)  XrmStringToQuarkList(str, name)
#define XrmStringToClass(class)         XrmStringToQuark(class)
#define XrmStringToClassList(str,class) XrmStringToQuarkList(str, class)
#define XrmStringToRepresentation(string) \
                XrmStringToQuark(string)

typedef struct {
        unsigned int            size;
        XPointer                addr;
} XrmValue, *XrmValuePtr;

typedef void                    *XrmHashBucket;
typedef XrmHashBucket           *XrmHashTable;
typedef XrmHashTable            XrmSearchList[];
typedef void                    *XrmDatabase;

#define XrmEnumAllLevels        0
#define XrmEnumOneLevel         1
```

Figure 6-131: `<X11/Xresource.h>`**, Part 2 of 2**

```
typedef enum {
        XrmoptionNoArg,
        XrmoptionIsArg,
        XrmoptionStickyArg,
        XrmoptionSepArg,
        XrmoptionResArg,
        XrmoptionSkipArg,
        XrmoptionSkipLine,
        XrmoptionSkipNArgs
} XrmOptionKind;

typedef struct {
        char                    *option;
        char                    *specifier;
        XrmOptionKind           argKind;
        XPointer                value;
} XrmOptionDescRec, *XrmOptionDescList;
```

Figure 6-132: `<X11/Xutil.h>`**, Part 1 of 5**

```
#define NoValue         0x0000
#define XValue          0x0001
#define YValue          0x0002
#define WidthValue              0x0004
#define HeightValue     0x0008
#define AllValues               0x000F
#define XNegative               0x0010
#define YNegative               0x0020

typedef struct {
        long flags;
        int x, y;
        int width, height;
        int min_width, min_height;
        int max_width, max_height;
        int width_inc, height_inc;
        struct {
                int x;
                int y;
        } min_aspect, max_aspect;
        int base_width, base_height;
        int win_gravity;
} XSizeHints;

#define USPosition              (1L << 0)
#define USSize          (1L << 1)
#define PPosition               (1L << 2)
#define PSize           (1L << 3)
#define PMinSize        (1L << 4)
#define PMaxSize        (1L << 5)
#define PResizeInc              (1L << 6)
#define PAspect         (1L << 7)
#define PBaseSize               (1L << 8)
#define PWinGravity             (1L << 9)
#define PAllHints (PPosition|PSize|PMinSize|PMaxSize|PResizeInc|PAspect)
```

Figure 6-133: `<X11/Xutil.h>`**, Part 2 of 5**

```
typedef struct {
        long    flags;
        Bool    input;
        int     initial_state;
        Pixmap  icon_pixmap;
        Window  icon_window;
        int     icon_x, icon_y;
        Pixmap  icon_mask;
        XID     window_group;
} XWMHints;

#define InputHint               (1L << 0)
#define StateHint               (1L << 1)
#define IconPixmapHint   (1L << 2)
#define IconWindowHint   (1L << 3)
#define IconPositionHint (1L << 4)
#define IconMaskHint     (1L << 5)
#define WindowGroupHint  (1L << 6)
#define AllHints (InputHint|StateHint|
        IconPixmapHint|IconWindowHint|
        IconPositionHint|IconMaskHint|WindowGroupHint)

#define WithdrawnState          0
#define NormalState             1
#define IconicState             3

typedef struct {
        unsigned char           *value;
        Atom                    encoding;
        int                     format;
        unsigned long           nitems;
} XTextProperty;

#define XNoMemory                       -1
#define XLocaleNotSupported             -2
#define XConverterNotFound              -3
```

Figure 6-134: `<X11/Xutil.h>`**, Part 3 of 5**

```
typedef int XContext;

typedef enum {
        XStringStyle,
        XCompoundTextStyle,
        XTextStyle,
        XStdICCTextStyle
} XICCEncodingStyle;

typedef struct {
        int min_width, min_height;
        int max_width, max_height;
        int width_inc, height_inc;
} XIconSize;

typedef struct {
        char *res_name;
        char *res_class;
} XClassHint;

#define XDestroyImage(ximage)
        ((*((ximage)->f.destroy_image))((ximage)))
#define XGetPixel(ximage, x, y)
        ((*((ximage)->f.get_pixel))((ximage), (x), (y)))
#define XPutPixel(ximage, x, y, pixel)
        ((*((ximage)->f.put_pixel))((ximage), (x), (y), (pixel)))
#define XSubImage(ximage, x, y, width, height)
        ((*((ximage)->f.sub_image))((ximage), (x), (y), (width), (height)))
#define XAddPixel(ximage, value)
        ((*((ximage)->f.add_pixel))((ximage), (value)))

typedef struct _XComposeStatus {
        XPointer compose_ptr;
        int chars_matched;
} XComposeStatus;
```

Figure 6-135: `<X11/Xutil.h>`**, Part 4 of 5**

```
#define IsKeypadKey(keysym)
        (((unsigned)(keysym) >= XK_KP_Space) && ((unsigned)(keysym) <= XK_KP_Equal))
#define IsCursorKey(keysym)
        (((unsigned)(keysym) >= XK_Home) && ((unsigned)(keysym) < XK_Select))
#define IsPFKey(keysym)
        (((unsigned)(keysym) >= XK_KP_F1) && ((unsigned)(keysym) <= XK_KP_F4))
#define IsFunctionKey(keysym)
        (((unsigned)(keysym) >= XK_F1) && ((unsigned)(keysym) <= XK_F35))
#define IsMiscFunctionKey(keysym)
        (((unsigned)(keysym) >= XK_Select) && ((unsigned)(keysym) <= XK_Break))
#define IsModifierKey(keysym)
        ((((unsigned)(keysym) >= XK_Shift_L) && ((unsigned)(keysym) <= XK_Hyper_R))
        || ((unsigned)(keysym) == XK_Mode_switch)
        || ((unsigned)(keysym) == XK_Num_Lock))

typedef void *Region;

#define RectangleOut    0
#define RectangleIn             1
#define RectanglePart   2

typedef struct {
        Visual *visual;
        VisualID visualid;
        int     screen;
        int     depth;
        int     class;
        unsigned long red_mask;
        unsigned long green_mask;
        unsigned long blue_mask;
        int     colormap_size;
        int     bits_per_rgb;
} XVisualInfo;
```

Figure 6-136: `<X11/Xutil.h>`**, Part 5 of 5**

```
#define VisualNoMask            0x0
#define VisualIDMask            0x1
#define VisualScreenMask        0x2
#define VisualDepthMask         0x4
#define VisualClassMask         0x8
#define VisualRedMaskMask               0x10
#define VisualGreenMaskMask             0x20
#define VisualBlueMaskMask              0x40
#define VisualColormapSizeMask  0x80
#define VisualBitsPerRGBMask            0x100
#define VisualAllMask           0x1FF

typedef struct {
        Colormap                colormap;
        unsigned long           red_max;
        unsigned long           red_mult;
        unsigned long           green_max;
        unsigned long           green_mult;
        unsigned long           blue_max;
        unsigned long           blue_mult;
        unsigned long           base_pixel;
        VisualID                visualid;
        XID                     killid;
} XStandardColormap;

#define ReleaseByFreeingColormap ((XID) 1L)
#define BitmapSuccess           0
#define BitmapOpenFailed        1
#define BitmapFileInvalid               2
#define BitmapNoMemory          3
#define XCSUCCESS                       0
#define XCNOMEM                 1
#define XCNOENT                 2
```

TCP/IP Data Definitions

NOTE This section is new, but will not be diffmarked.

This section contains standard data definitions that describe system data for the optional TCP / IP Interfaces. These data definitions are referred to by their names in angle brackets: <*name.h*> and <sys / *name.h*>. Included in these data definitions are macro definitions and structure definitions. While an ABI-conforming system may provide TCP / IP interfaces, it need not contain the actual data definitions referenced here. Programmers should observe that the sources of the structures defined in these data definitions are defined in SVID.

Figure 6-137: `<netinet/in.h>`

```
#define   INADDR_ANY       (u_long)0x00000000
#define   INADDR_LOOPBACK  (u_long)0x7F000001
#define   INADDR_BROADCAST (u_long)0xffffffff
#define   IPPROTO_TCP      6
#define   IPPROTO_IP       0
#define   IP_OPTIONS       1

struct in_addr {
      union {
            struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;
            struct { u_short s_w1,s_w2; } S_un_w;
            u_long S_addr;
      } S_un;

#define IN_SET_LOOPBACK_ADDR(a)  {(a)->sin_addr.s_addr=htonl(INADDR_LOOPBACK);

struct sockaddr_in {
      short    sin_family;
      u_short  sin_port;
      struct   in_addr sin_addr;
      char     sin_zero[8];
};
```

Figure 6-138: `<netinet/ip.h>`

```
#define IPOPT_EOL   0
#define IPOPT_NOP   1
#define IPOPT_LSRR  131
#define IPOPT_SSRR  137
```

Figure 6-139: `<netinet/tcp.h>`

```
#define TCP_NODELAY     0x01
```

7 DEVELOPMENT ENVIRONMENT

Development Commands

NOTE

THE FACILITIES AND INTERFACES DESCRIBED IN THIS SECTION ARE OPTIONAL COMPONENTS OF THE *System V Application Binary Interface*. The information in this chapter corresponds to Chapter 11 of the generic *System V Application Binary Interface*.

The Development Environment for SPARC implementations of System V Release 4.2 will contain all of the development commands required by the System V ABI, namely;

`as`	`cc`	`ld`
`m4`	`lex`	`yacc`

Each command accepts all of the options required by the System V ABI, as defined in the SD_CMD section of the *System V Interface Definition, Third Edition*.

PATH Access to Development Tools

The development environment for the SPARC System V implementations is accessible using the system default value for PATH. The default if no options are given to the `cc` command is to use the libraries and object file formats that are required for ABI compliance.

Software Packaging Tools

The development environment for SPARC implementations of the System V ABI shall include each of the following commands as defined in the AS_CMD section of *System V Interface Definition, Third Edition*.

`pkgproto   pkgtrans   pkgmk`

System Headers

Systems that do not have an ABI Development Environment may not have system header files. If an ABI Development Environment is supported, system header files will be included with the Development Environment. The primary source for contents of header files is always the *System V Interface Definition, Third Edition*. In those cases where SVID Third Edition doesn't specify the contents of system headers, Chapter 6 "Data Definitions" of this document shall define the associations of data elements to system headers for compilation. For greatest source portability, applications should only depend on header file contents defined in SVID.

Static Archives

Level 1 interfaces defined in *System V Interface Definition, Third Edition*, for each of the following libraries, may be statically linked safely into applications. The resulting executable will not be made non-compliant to the ABI solely because of the static linkage of such members in the executable.

`libm`

The archive `libm.a` is located in /usr/lib on conforming SPARC development environments.

8 EXECUTION ENVIRONMENT

Application Environment

This section specifies the execution environment information available to application programs running on an SPARC ABI-conforming computer.

The /dev Subtree

NOTE THE FACILITIES AND INTERFACES DESCRIBED IN THIS SECTION ARE OPTIONAL COMPONENTS OF THE *System V SPARC Application Binary Interface*.

All networking device files described in the Generic ABI shall be supported on all SPARC ABI-conforming computers. In addition, the following device files are required to be present on all SPARC ABI-conforming computers.

`/dev/null` This device file is a special ‘‘null’’ device that may be used to test programs of provide a data sink. This file is writable by all processes.

`/dev/tty` This device file is a special one that directs all output to the controlling TTY of the current process group. This file is readable and writable by all processes.

`/dev/sxtXX`
`/dev/ttyXX` These device files, where XX represents a two-digit integer, represent device entries for terminal sessions. All these device files must be examined by the `ttyname()` call. Applications must not have the device names of individual terminals hard-coded within them. The `sxt` entries are optional in the system but, if present must be included in the library routine’s search.

`/dev/dsk/`
`/dev/rdsk/` These directories contain the raw and block disk device files. They are of the form:

c#t#d#s#

where ’c’ is followed by a controller number,
’t’ is followed by a target number,
’d’ is followed by a disk unit number,
’s’ is followed by a disk slice number.

IN Index

Index

C

D

E

P

Q

R

S