SYSTEM V
APPLICATION
BINARY INTERFACE

SPARC® Processor
Supplement

Third Edition
## Contents

### 1 INTRODUCTION
- SPARC Processor and the System V ABI 1-1
- How to Use the SPARC ABI Supplement 1-2

### 2 SOFTWARE INSTALLATION
- Software Distribution Formats 2-1

### 3 LOW-LEVEL SYSTEM INFORMATION
- Machine Interface 3-1
- Function Calling Sequence 3-8
- Operating System Interface 3-21
- Coding Examples 3-34

### 4 OBJECT FILES
- ELF Header 4-1
- Sections 4-2
- Relocation 4-3

### 5 PROGRAM LOADING AND DYNAMIC LINKING
- Program Loading 5-1
- Dynamic Linking 5-5

### 6 LIBRARIES
- System Library 6-1
- System Data Interfaces 6-11

### 7 DEVELOPMENT ENVIRONMENT
- Development Commands 7-1
- Software Packaging Tools 7-2
## Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 EXECUTION ENVIRONMENT</td>
<td></td>
</tr>
<tr>
<td>Application Environment</td>
<td>8-1</td>
</tr>
<tr>
<td>Index</td>
<td></td>
</tr>
<tr>
<td>Index</td>
<td>IN-1</td>
</tr>
</tbody>
</table>

---

**SPARC™ PROCESSOR SUPPLEMENT**
Figures and Tables

Figure 3-1: Scalar Types
Figure 3-2: Structure Smaller Than a Word
Figure 3-3: No Padding
Figure 3-4: Internal Padding
Figure 3-5: Internal and Tail Padding
Figure 3-6: union Allocation
Figure 3-7: Bit-Field Ranges
Figure 3-8: Bit Numbering
Figure 3-9: Left-to-Right Allocation
Figure 3-10: Boundary Alignment
Figure 3-11: Storage Unit Sharing
Figure 3-12: union Allocation
Figure 3-13: Unnamed Bit-Fields
Figure 3-14: A Function's Window Registers
Figure 3-15: A Function's Global Registers
Figure 3-16: Standard Stack Frame
Figure 3-17: Function Prologue
Figure 3-18: Register Windows
Figure 3-19: Integral and Pointer Arguments
Figure 3-20: Floating-Point Arguments
Figure 3-21: Sending Structure, Union, and Quad-Precision Arguments
Figure 3-22: Receiving Structure, Union, and Quad-Precision Arguments
Figure 3-23: Function Epilogue
Figure 3-24: Alternative Function Epilogue
Figure 3-25: Function Epilogue
Figure 3-26: Virtual Address Configuration
Figure 3-27: Hardware Traps and Signals
Figure 3-28: Software Trap Types
Figure 3-29: Declaration for main
Figure 3-30: Processor State Register Fields
Figure 3-31: Floating-Point State Register Fields
Figure 3-32: Initial Process Stack
Figure 3-33: Auxiliary Vector
Figure 3-34: Auxiliary Vector Types, a_type
Figure 3-35: Example Process Stack
Figure 3-36: Position-Independent Function Prologue
Figure 3-37: Absolute Load and Store
Figure 3-38: Small Model Position-Independent Load and Store
Figure 3-39: Large Model Position-Independent Load and Store
Figure 3-40: Direct Function Call, All Models
Figure 3-41: Absolute Indirect Function Call
Figure 3-42: Small Model Position-Independent Indirect Function Call
Figure 3-43: Large Model Position-Independent Indirect Function Call
Figure 3-44: Branch Instruction, All Models
Figure 3-45: Absolute switch Code
Figure 3-46: Position-Independent switch Code
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-47</td>
<td>C Stack Frame</td>
<td>3-44</td>
</tr>
<tr>
<td>3-48</td>
<td>Argument Stack Positions</td>
<td>3-45</td>
</tr>
<tr>
<td>3-49</td>
<td>Dynamic Stack Allocation</td>
<td>3-47</td>
</tr>
<tr>
<td>4-1</td>
<td>SPARC Identification, e_ident</td>
<td>4-1</td>
</tr>
<tr>
<td>4-2</td>
<td>Special Sections</td>
<td>4-2</td>
</tr>
<tr>
<td>4-3</td>
<td>Relocatable Fields</td>
<td>4-3</td>
</tr>
<tr>
<td>4-4</td>
<td>Relocation Types</td>
<td>4-5</td>
</tr>
<tr>
<td>5-1</td>
<td>Executable File</td>
<td>5-1</td>
</tr>
<tr>
<td>5-2</td>
<td>Program Header Segments</td>
<td>5-2</td>
</tr>
<tr>
<td>5-3</td>
<td>Process Image Segments</td>
<td>5-3</td>
</tr>
<tr>
<td>5-4</td>
<td>Example Shared Object Segment Addresses</td>
<td>5-4</td>
</tr>
<tr>
<td>5-5</td>
<td>Global Offset Table</td>
<td>5-6</td>
</tr>
<tr>
<td>5-6</td>
<td>Procedure Linkage Table Example</td>
<td>5-8</td>
</tr>
<tr>
<td>6-1</td>
<td>libsys Support Routines</td>
<td>6-1</td>
</tr>
<tr>
<td>6-2</td>
<td>libsys, Global External Data Symbols</td>
<td>6-10</td>
</tr>
<tr>
<td>6-3</td>
<td>&lt;assert.h&gt;</td>
<td>6-11</td>
</tr>
<tr>
<td>6-4</td>
<td>&lt;ctype.h&gt;</td>
<td>6-12</td>
</tr>
<tr>
<td>6-5</td>
<td>&lt;dirent.h&gt;</td>
<td>6-13</td>
</tr>
<tr>
<td>6-6</td>
<td>&lt;errno.h&gt;</td>
<td>6-13</td>
</tr>
<tr>
<td>6-7</td>
<td>&lt;fcntl.h&gt;</td>
<td>6-16</td>
</tr>
<tr>
<td>6-8</td>
<td>&lt;float.h&gt;</td>
<td>6-17</td>
</tr>
<tr>
<td>6-9</td>
<td>&lt;fmtmsg.h&gt;</td>
<td>6-17</td>
</tr>
<tr>
<td>6-10</td>
<td>&lt;ftw.h&gt;</td>
<td>6-18</td>
</tr>
<tr>
<td>6-11</td>
<td>&lt;grp.h&gt;</td>
<td>6-18</td>
</tr>
<tr>
<td>6-12</td>
<td>&lt;sys/ipc.h&gt;</td>
<td>6-19</td>
</tr>
<tr>
<td>6-13</td>
<td>&lt;langinfo.h&gt;</td>
<td>6-19</td>
</tr>
<tr>
<td>6-14</td>
<td>&lt;limits.h&gt;</td>
<td>6-21</td>
</tr>
<tr>
<td>6-15</td>
<td>&lt;locale.h&gt;</td>
<td>6-22</td>
</tr>
<tr>
<td>6-16</td>
<td>&lt;math.h&gt;</td>
<td>6-22</td>
</tr>
<tr>
<td>6-17</td>
<td>&lt;sys/rman.h&gt;</td>
<td>6-23</td>
</tr>
<tr>
<td>6-18</td>
<td>&lt;sys/mount.h&gt;</td>
<td>6-23</td>
</tr>
<tr>
<td>6-19</td>
<td>&lt;sys/msg.h&gt;</td>
<td>6-24</td>
</tr>
<tr>
<td>6-20</td>
<td>&lt;netconfig.h&gt;</td>
<td>6-25</td>
</tr>
<tr>
<td>6-21</td>
<td>&lt;netdir.h&gt;</td>
<td>6-26</td>
</tr>
<tr>
<td>6-22</td>
<td>&lt;nl_types.h&gt;</td>
<td>6-27</td>
</tr>
<tr>
<td>6-23</td>
<td>&lt;sys/param.h&gt;</td>
<td>6-27</td>
</tr>
<tr>
<td>6-24</td>
<td>&lt;poll.h&gt;</td>
<td>6-28</td>
</tr>
<tr>
<td>6-25</td>
<td>&lt;sys/procset.h&gt;</td>
<td>6-29</td>
</tr>
<tr>
<td>6-26</td>
<td>&lt;pwd.h&gt;</td>
<td>6-30</td>
</tr>
<tr>
<td>6-27</td>
<td>&lt;sys/resource.h&gt;</td>
<td>6-30</td>
</tr>
<tr>
<td>6-28</td>
<td>&lt;rpc.h&gt;</td>
<td>6-31</td>
</tr>
<tr>
<td>6-29</td>
<td>&lt;search.h&gt;</td>
<td>6-38</td>
</tr>
<tr>
<td>6-30</td>
<td>&lt;sys/sem.h&gt;</td>
<td>6-39</td>
</tr>
<tr>
<td>6-31</td>
<td>&lt;setjmp.h&gt;</td>
<td>6-39</td>
</tr>
<tr>
<td>6-32</td>
<td>&lt;sys/shm.h&gt;</td>
<td>6-40</td>
</tr>
<tr>
<td>6-33</td>
<td>&lt;signal.h&gt;</td>
<td>6-40</td>
</tr>
<tr>
<td>6-34</td>
<td>&lt;sys/siginfo.h&gt;</td>
<td>6-42</td>
</tr>
<tr>
<td>6-35</td>
<td>&lt;sys/stat.h&gt;</td>
<td>6-43</td>
</tr>
<tr>
<td>6-36</td>
<td>&lt;sys/statvfs.h&gt;</td>
<td>6-45</td>
</tr>
<tr>
<td>6-37</td>
<td>&lt;stdarg.h&gt;</td>
<td>6-45</td>
</tr>
<tr>
<td>6-38</td>
<td>&lt;stddef.h&gt;</td>
<td>6-46</td>
</tr>
<tr>
<td>6-39</td>
<td>&lt;stdio.h&gt;</td>
<td>6-47</td>
</tr>
</tbody>
</table>
Figure 6-40: <stdlib.h> 6-48
Figure 6-41: <stropts.h> 6-49
Figure 6-42: <termios.h> 6-51
Figure 6-43: <sys/ticltls.h> 6-55
Figure 6-44: <sys/ticots.h> 6-55
Figure 6-45: <sys/ticotsord.h> 6-55
Figure 6-46: <sys/tihdr.h> 6-56
Figure 6-47: <sys/time.h> 6-57
Figure 6-48: <sys/times.h> 6-58
Figure 6-49: <sys/timod.h> 6-58
Figure 6-50: <sys/tiuser.h>, Service Types 6-58
Figure 6-51: <tiuser.h>, Transport Interface States 6-59
Figure 6-52: <sys/tiuser.h>, User-level Events 6-59
Figure 6-53: <sys/tiuser.h>, Error Return Values 6-60
Figure 6-54: <sys/tiuser.h>, Transport Interface Data Structures 6-60
Figure 6-55: <sys/tiuser.h>, Structure Types 6-62
Figure 6-56: <sys/tiuser.h>, Fields of Structures 6-62
Figure 6-57: <sys/tiuser.h>, Events Bitmasks 6-62
Figure 6-58: <sys/tiuser.h>, Flags 6-63
Figure 6-59: <sys/types.h> 6-63
Figure 6-60: <ucontext.h> 6-64
Figure 6-61: <sys/uio.h> 6-65
Figure 6-62: <ulimit.h> 6-65
Figure 6-63: <unistd.h> 6-66
Figure 6-64: <utime.h> 6-67
Figure 6-65: <sys/utsname.h> 6-67
Figure 6-66: <wait.h> 6-68
Figure 6-67: <X11/Composite.h> 6-70
Figure 6-68: <X11/Constraint.h> 6-70
Figure 6-69: <X11/Core.h> 6-70
Figure 6-70: <X11/cursorfont.h>, Part 1 of 3 6-71
Figure 6-71: <X11/cursorfont.h>, Part 2 of 3 6-72
Figure 6-72: <X11/cursorfont.h>, Part 3 of 3 6-73
Figure 6-73: <X11/Intrinsic.h>, Part 1 of 6 6-74
Figure 6-74: <X11/Intrinsic.h>, Part 2 of 6 6-75
Figure 6-75: <X11/Intrinsic.h>, Part 3 of 6 6-76
Figure 6-76: <X11/Intrinsic.h>, Part 4 of 6 6-77
Figure 6-77: <X11/Intrinsic.h>, Part 5 of 6 6-78
Figure 6-78: <X11/Intrinsic.h>, Part 6 of 6 6-79
Figure 6-79: <X11/Object.h> 6-79
Figure 6-80: <X11/RectObj.h> 6-79
Figure 6-81: <X11/Shell.h> 6-80
Figure 6-82: <X11/Vendor.h> 6-80
Figure 6-83: <X11/X.h>, Part 1 of 12 6-81
Figure 6-84: <X11/X.h>, Part 2 of 12 6-82
Figure 6-85: <X11/X.h>, Part 3 of 12 6-83
Figure 6-86: <X11/X.h>, Part 4 of 12 6-84
Figure 6-87: <X11/X.h>, Part 5 of 12 6-85
Figure 6-88: <X11/X.h>, Part 6 of 12 6-86
Figure 6-89: <X11/X.h>, Part 7 of 12 6-87
Figure 6-90: <X11/X.h>, Part 8 of 12 6-88
Figure 6-91: <X11/X.h>, Part 9 of 12 6-89
Figure 6-92: <X11/X.h>, Part 10 of 12 6-90
Figure 6-93: <X11/X.h>, Part 11 of 12 6-91
Figure 6-94: <X11/X.h>, Part 12 of 12 6-92
Figure 6-95: <X11/Xatom.h>, Part 1 of 3 6-93
Figure 6-96: <X11/Xatom.h>, Part 2 of 3 6-94
Figure 6-97: <X11/Xatom.h>, Part 3 of 3 6-95
Figure 6-98: <X11/Xcms.h>, Part 1 of 5 6-96
Figure 6-99: <X11/Xcms.h>, Part 2 of 5 6-97
Figure 6-100: <X11/Xcms.h>, Part 3 of 5 6-98
Figure 6-101: <X11/Xcms.h>, Part 4 of 5 6-99
Figure 6-102: <X11/Xcms.h>, Part 5 of 5 6-100
Figure 6-103: <X11/Xlib.h> Part 1 of 27 6-101
Figure 6-104: <X11/Xlib.h> Part 2 of 27 6-101
Figure 6-105: <X11/Xlib.h> Part 3 of 27 6-102
Figure 6-106: <X11/Xlib.h> Part 4 of 27 6-103
Figure 6-107: <X11/Xlib.h> Part 5 of 27 6-104
Figure 6-108: <X11/Xlib.h> Part 6 of 27 6-105
Figure 6-109: <X11/Xlib.h> Part 7 of 27 6-106
Figure 6-110: <X11/Xlib.h> Part 8 of 27 6-107
Figure 6-111: <X11/Xlib.h> Part 9 of 27 6-108
Figure 6-112: <X11/Xlib.h> Part 10 of 27 6-109
Figure 6-113: <X11/Xlib.h> Part 11 of 27 6-110
Figure 6-114: <X11/Xlib.h> Part 12 of 27 6-111
Figure 6-115: <X11/Xlib.h> Part 13 of 27 6-112
Figure 6-116: <X11/Xlib.h> Part 14 of 27 6-113
Figure 6-117: <X11/Xlib.h> Part 15 of 27 6-114
Figure 6-118: <X11/Xlib.h> Part 16 of 27 6-115
Figure 6-119: <X11/Xlib.h> Part 17 of 27 6-116
Figure 6-120: <X11/Xlib.h> Part 18 of 27 6-117
Figure 6-121: <X11/Xlib.h> Part 19 of 27 6-118
Figure 6-122: <X11/Xlib.h> Part 20 of 27 6-119
Figure 6-123: <X11/Xlib.h> Part 21 of 27 6-120
Figure 6-124: <X11/Xlib.h> Part 22 of 27 6-121
Figure 6-125: <X11/Xlib.h> Part 23 of 27 6-122
Figure 6-126: <X11/Xlib.h> Part 24 of 27 6-123
Figure 6-127: <X11/Xlib.h> Part 25 of 27 6-124
Figure 6-128: <X11/Xlib.h> Part 26 of 27 6-125
Figure 6-129: <X11/Xlib.h> Part 27 of 27 6-126
Figure 6-130: <X11/Xresource.h>, Part 1 of 2 6-127
Figure 6-131: <X11/Xresource.h>, Part 2 of 2 6-128
Figure 6-132: <X11/Xutil.h>, Part 1 of 5 6-129
Figure 6-133: <X11/Xutil.h>, Part 2 of 5 6-130
Figure 6-134: <X11/Xutil.h>, Part 3 of 5 6-131
Figure 6-135: <X11/Xutil.h>, Part 4 of 5 6-132
Figure 6-136: <X11/Xutil.h>, Part 5 of 5 6-133
Figure 6-137: <netinet/in.h> 6-135
Figure 6-138: <netinet/ip.h> 6-135
Figure 6-139: <netinet/tcp.h> 6-135
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 © 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

Table of Contents
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 3-1
Processor Architecture 3-1
Data Representation 3-1
  • Fundamental Types 3-1
  • Aggregates and Unions 3-3
  • Bit-Fields 3-5

Function Calling Sequence 3-8
Registers and the Stack Frame 3-8
Integral and Pointer Arguments 3-15
Floating-Point Arguments 3-15
Structure, Union, and Quad-Precision Arguments 3-16
Functions Returning Scalars or No Value 3-17
Functions Returning Structures, Unions, or Quad-Precision Values 3-18

Operating System Interface 3-21
Virtual Address Space 3-21
  • Page Size 3-21
  • Virtual Address Assignments 3-21
  • Managing the Process Stack 3-23
  • Coding Guidelines 3-23
Trap Interface 3-24
  • Hardware Trap Types 3-24
  • Software Trap Types 3-25
Process Initialization 3-27
  • Special Registers 3-27
  • Process Stack and Registers 3-29

Coding Examples 3-34
Code Model Overview 3-35
Position-Independent Function Prologue 3-36
Data Objects 3-37
Function Calls 3-39
Branching 3-42
C Stack Frame 3-44
Variable Argument List 3-44
Allocating Stack Space Dynamically 3-45
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.
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**

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

Byte aligned, sizeof is 1

**Figure 3-3: No Padding**

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

Word aligned, sizeof is 8

**Figure 3-4: Internal Padding**

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

Halfword aligned, sizeof is 4
Bit-Fields

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

<table>
<thead>
<tr>
<th>Bit-field Type</th>
<th>Width ( w )</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>signed char</td>
<td>1 to 8</td>
<td>(-2^{w-1} ) to (2^{w-1}-1)</td>
</tr>
<tr>
<td>char</td>
<td></td>
<td>0 to (2^w-1)</td>
</tr>
<tr>
<td>unsigned char</td>
<td></td>
<td>0 to (2^w-1)</td>
</tr>
<tr>
<td>signed short</td>
<td>1 to 16</td>
<td>(-2^{w-1} ) to (2^{w-1}-1)</td>
</tr>
<tr>
<td>short</td>
<td></td>
<td>0 to (2^w-1)</td>
</tr>
<tr>
<td>unsigned short</td>
<td></td>
<td>0 to (2^w-1)</td>
</tr>
<tr>
<td>signed int</td>
<td>1 to 32</td>
<td>(-2^{w-1} ) to (2^{w-1}-1)</td>
</tr>
<tr>
<td>int</td>
<td></td>
<td>0 to (2^w-1)</td>
</tr>
<tr>
<td>enum</td>
<td></td>
<td>0 to (2^w-1)</td>
</tr>
<tr>
<td>unsigned int</td>
<td></td>
<td>0 to (2^w-1)</td>
</tr>
</tbody>
</table>
“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 3-10: Boundary Alignment

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

Word aligned, sizeof is 12

```
+---+---+---+---+---+
| s | j | pad | t | pad |
+---+---+---+---+---+
| pad | u | pad |
+---+---+---+---+---+
```

Figure 3-11: Storage Unit Sharing

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

Halfword aligned, sizeof is 2

```
+---+---+
| c | s |
+---+---+
```

Figure 3-12: union Allocation

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

Halfword aligned, sizeof is 2

```
+---+---+
| c | pad |
+---+---+
| s | pad |
+---+---+
```

Figure 3-13: Unnamed Bit-Fields

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

Byte aligned, sizeof is 9

```
+---+---+
| c | :0 |
+---+---+
| pad | :9 |
+---+---+
| pad |
+---+---+
| e |  |
+---+---+
```

SPARC™ PROCESSOR SUPPLEMENT
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.

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

NOTE

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>%17 %r31</code></td>
<td>return address – 8 †</td>
</tr>
<tr>
<td></td>
<td><code>%16 %r30</code></td>
<td>frame pointer †</td>
</tr>
<tr>
<td></td>
<td><code>%15 %r29</code></td>
<td>incoming param 5 †</td>
</tr>
<tr>
<td></td>
<td><code>%14 %r28</code></td>
<td>incoming param 4 †</td>
</tr>
<tr>
<td></td>
<td><code>%13 %r27</code></td>
<td>incoming param 3 †</td>
</tr>
<tr>
<td></td>
<td><code>%12 %r26</code></td>
<td>incoming param 2 †</td>
</tr>
<tr>
<td></td>
<td><code>%11 %r25</code></td>
<td>incoming param 1 †</td>
</tr>
<tr>
<td></td>
<td><code>%10 %r24</code></td>
<td>incoming param 0, †</td>
</tr>
<tr>
<td>in</td>
<td><code>%17 %r23</code></td>
<td>local 7 †</td>
</tr>
<tr>
<td></td>
<td><code>%16 %r22</code></td>
<td>local 6 †</td>
</tr>
<tr>
<td></td>
<td><code>%15 %r21</code></td>
<td>local 5 †</td>
</tr>
<tr>
<td></td>
<td><code>%14 %r20</code></td>
<td>local 4 †</td>
</tr>
<tr>
<td></td>
<td><code>%13 %r19</code></td>
<td>local 3 †</td>
</tr>
<tr>
<td></td>
<td><code>%12 %r18</code></td>
<td>local 2 †</td>
</tr>
<tr>
<td></td>
<td><code>%11 %r17</code></td>
<td>local 1 †</td>
</tr>
<tr>
<td></td>
<td><code>%10 %r16</code></td>
<td>local 0 †</td>
</tr>
<tr>
<td></td>
<td><code>%o7 %r15</code></td>
<td>address of call instruction, ‡</td>
</tr>
<tr>
<td></td>
<td><code>%o6 %r14</code></td>
<td>temporary value</td>
</tr>
<tr>
<td>local</td>
<td><code>%o5 %r13</code></td>
<td>stack pointer †</td>
</tr>
<tr>
<td></td>
<td><code>%o4 %r12</code></td>
<td>outgoing param 5 ‡</td>
</tr>
<tr>
<td></td>
<td><code>%o3 %r11</code></td>
<td>outgoing param 4 ‡</td>
</tr>
<tr>
<td></td>
<td><code>%o2 %r10</code></td>
<td>outgoing param 3 ‡</td>
</tr>
<tr>
<td></td>
<td><code>%o1 %r9</code></td>
<td>outgoing param 1 ‡</td>
</tr>
<tr>
<td></td>
<td><code>%o0 %r8</code></td>
<td>outgoing param 0 ‡</td>
</tr>
<tr>
<td></td>
<td></td>
<td>incoming return value</td>
</tr>
</tbody>
</table>

### Figure 3-15: A Function’s Global Registers

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>%g7 %r7</code></td>
<td>global 7 (reserved for system)</td>
</tr>
<tr>
<td></td>
<td><code>%g6 %r6</code></td>
<td>global 6 (reserved for system)</td>
</tr>
<tr>
<td></td>
<td><code>%g5 %r5</code></td>
<td>global 5 (reserved for system)</td>
</tr>
<tr>
<td></td>
<td><code>%g4 %r4</code></td>
<td>global 4 (reserved for application)</td>
</tr>
<tr>
<td></td>
<td><code>%g3 %r3</code></td>
<td>global 3 (reserved for application)</td>
</tr>
<tr>
<td></td>
<td><code>%g2 %r2</code></td>
<td>global 2 (reserved for application)</td>
</tr>
<tr>
<td></td>
<td><code>%g1 %r1</code></td>
<td>global 1 ‡</td>
</tr>
</tbody>
</table>
Figure 3-15: A Function’s Global Registers (continued)

<table>
<thead>
<tr>
<th></th>
<th>%g0</th>
<th>%r0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>global</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>%f31</td>
<td></td>
<td>floating-point value †</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>⋮</td>
<td>⋮</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>%f0</td>
<td></td>
<td>floating-point value, ‡</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>floating-point return value</td>
</tr>
<tr>
<td>special</td>
<td>%y</td>
<td></td>
<td>Y register ‡</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Base</th>
<th>Offset</th>
<th>Contents</th>
<th>Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>%fp</td>
<td>+92</td>
<td>unspecified</td>
<td>High addresses</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>variable size</td>
<td></td>
</tr>
<tr>
<td>%fp</td>
<td>+68</td>
<td>incoming arguments 6, ...</td>
<td>Previous</td>
</tr>
<tr>
<td>%fp</td>
<td>+64</td>
<td>six words into which function may write incoming arguments 0 to 5</td>
<td></td>
</tr>
</tbody>
</table>
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.

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

%10 through %15  *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 \textit{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**

<table>
<thead>
<tr>
<th>Call</th>
<th>Argument</th>
<th>Caller</th>
<th>Callee</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>%o0</td>
<td>%i0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>%o1</td>
<td>%i1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>%o2</td>
<td>%i2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>%o3</td>
<td>%i3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>%o4</td>
<td>%i4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>%o5</td>
<td>%i5</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>%sp+92</td>
<td>%fp+92</td>
<td></td>
</tr>
</tbody>
</table>

Floating-Point Arguments

The integer \textit{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**

<table>
<thead>
<tr>
<th>Call</th>
<th>Argument</th>
<th>Caller</th>
<th>Callee</th>
</tr>
</thead>
<tbody>
<tr>
<td>h(1.414, 1, 2.998e10, 2.718);</td>
<td>word 0, 1.414</td>
<td>%o0</td>
<td>%i0</td>
</tr>
<tr>
<td></td>
<td>word 1, 1.414</td>
<td>%o1</td>
<td>%i1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>%o2</td>
<td>%i2</td>
</tr>
<tr>
<td></td>
<td>word 0, 2.998e10</td>
<td>%o3</td>
<td>%i3</td>
</tr>
<tr>
<td></td>
<td>word 1, 2.998e10</td>
<td>%o4</td>
<td>%i4</td>
</tr>
<tr>
<td></td>
<td>word 0, 2.718</td>
<td>%o5</td>
<td>%i5</td>
</tr>
<tr>
<td></td>
<td>word 1, 2.718</td>
<td>%sp+92</td>
<td>%fp+92</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Source</th>
<th>Compiler's Internal Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>caller()</td>
<td>caller()</td>
</tr>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>struct s s;</td>
<td>struct s, s2;</td>
</tr>
<tr>
<td>callee(s);</td>
<td>s2 = s;</td>
</tr>
<tr>
<td>}</td>
<td>callee(&amp;s2);</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Source</th>
<th>Compiler's Internal Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>callee(struct s arg)</td>
<td>callee(struct s *arg)</td>
</tr>
<tr>
<td>{</td>
<td>{</td>
</tr>
<tr>
<td>struct s, s2;</td>
<td>struct s, s2;</td>
</tr>
<tr>
<td>s.m = arg.m;</td>
<td>s.m = arg-&gt;m;</td>
</tr>
<tr>
<td>s2 = arg;</td>
<td>s2 = *arg;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>
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.
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](image)

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 map(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 0xfffffff. 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 map(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 map(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 map(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

<table>
<thead>
<tr>
<th>Trap Name</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>cp_disabled</td>
<td>SIGILL</td>
</tr>
<tr>
<td>cp_exception</td>
<td>SIGILL</td>
</tr>
<tr>
<td>data_access_error</td>
<td>unspecified</td>
</tr>
<tr>
<td>data_access_exception</td>
<td>SIGSEGV, SIGBUS</td>
</tr>
<tr>
<td>data_store_error</td>
<td>unspecified</td>
</tr>
<tr>
<td>division_by_zero</td>
<td>SIGFPE</td>
</tr>
<tr>
<td>fp_disabled</td>
<td>SIGILL</td>
</tr>
<tr>
<td>fp_exception</td>
<td>SIGFPE</td>
</tr>
<tr>
<td>illegal_instruction</td>
<td>SIGILL</td>
</tr>
<tr>
<td>instruction_access_exception</td>
<td>SIGSEGV, SIGBUS</td>
</tr>
<tr>
<td>mem_address_not_aligned</td>
<td>SIGBUS</td>
</tr>
<tr>
<td>privileged_instruction</td>
<td>SIGILL</td>
</tr>
<tr>
<td>r_register_access_error</td>
<td>unspecified</td>
</tr>
<tr>
<td>tag_overflow</td>
<td>SIGEMT</td>
</tr>
<tr>
<td>trap_instruction</td>
<td>see next table</td>
</tr>
<tr>
<td>window_overflow</td>
<td>none</td>
</tr>
<tr>
<td>window_underflow</td>
<td>none</td>
</tr>
</tbody>
</table>

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

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

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.

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.

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.
Operating System Interface

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

```c
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.

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>icc</td>
<td>unspecified</td>
<td>Integer condition codes unspecified</td>
</tr>
<tr>
<td>EC</td>
<td>unspecified</td>
<td>Coprocessor not specified</td>
</tr>
<tr>
<td>S</td>
<td>0</td>
<td>Processes run in user mode</td>
</tr>
<tr>
<td>ET</td>
<td>1</td>
<td>Traps enabled</td>
</tr>
</tbody>
</table>

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 rd asr and wr asr instructions, with the exceptions of rdy and wr y.

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.

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>RD</td>
<td>0</td>
<td>Round to nearest</td>
</tr>
<tr>
<td>TEM</td>
<td>0</td>
<td>Floating-point traps not enabled</td>
</tr>
<tr>
<td>NS</td>
<td>0</td>
<td>Nonstandard mode off</td>
</tr>
<tr>
<td>ftt</td>
<td>unspecified</td>
<td>Floating-point trap type unspecified</td>
</tr>
<tr>
<td>gne</td>
<td>0</td>
<td>Floating-point queue is empty</td>
</tr>
<tr>
<td>fcc</td>
<td>unspecified</td>
<td>Floating-point condition codes unspecified</td>
</tr>
<tr>
<td>aexc</td>
<td>0</td>
<td>No accrued exceptions</td>
</tr>
<tr>
<td>cexc</td>
<td>0</td>
<td>No current exceptions</td>
</tr>
</tbody>
</table>
Operating System Interface

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

<table>
<thead>
<tr>
<th>High addresses</th>
<th>Low addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unspecified</td>
<td>%sp+64</td>
</tr>
<tr>
<td>Information block, including</td>
<td>Argument count</td>
</tr>
<tr>
<td>argument strings</td>
<td></td>
</tr>
<tr>
<td>environment strings</td>
<td></td>
</tr>
<tr>
<td>auxiliary information</td>
<td>Window save area</td>
</tr>
<tr>
<td></td>
<td>(16 words)</td>
</tr>
<tr>
<td>Unspecified</td>
<td>%sp+0</td>
</tr>
<tr>
<td>Null auxiliary vector entry</td>
<td></td>
</tr>
<tr>
<td>Auxiliary vector</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>(2-word entries)</td>
<td></td>
</tr>
<tr>
<td>0 word</td>
<td></td>
</tr>
<tr>
<td>Environment pointers</td>
<td>Argument count</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>(one word each)</td>
<td></td>
</tr>
<tr>
<td>0 word</td>
<td></td>
</tr>
<tr>
<td>Argument pointers</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>(Argument count words)</td>
<td></td>
</tr>
<tr>
<td>%sp+64</td>
<td></td>
</tr>
<tr>
<td>%sp+0</td>
<td></td>
</tr>
</tbody>
</table>

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.

%f The system marks the deepest stack frame by setting the frame pointer to zero. No other frame’s %f has a zero value.
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**

```c
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**

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>a_un</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT_NULL</td>
<td>0</td>
<td>ignored</td>
</tr>
<tr>
<td>AT_IGNORE</td>
<td>1</td>
<td>ignored</td>
</tr>
<tr>
<td>AT_EXECFD</td>
<td>2</td>
<td>a_val</td>
</tr>
<tr>
<td>AT_PHDR</td>
<td>3</td>
<td>a_ptr</td>
</tr>
<tr>
<td>AT_PHENT</td>
<td>4</td>
<td>a_val</td>
</tr>
<tr>
<td>AT_PNUM</td>
<td>5</td>
<td>a_val</td>
</tr>
<tr>
<td>AT_PAGESZ</td>
<td>6</td>
<td>a_val</td>
</tr>
<tr>
<td>AT_BASE</td>
<td>7</td>
<td>a_ptr</td>
</tr>
<tr>
<td>AT_FLAGS</td>
<td>8</td>
<td>a_val</td>
</tr>
<tr>
<td>AT_ENTRY</td>
<td>9</td>
<td>a_ptr</td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7fffffff0</td>
<td>n: \0, pad</td>
</tr>
<tr>
<td></td>
<td>r: / bin</td>
</tr>
<tr>
<td></td>
<td>: / us</td>
</tr>
<tr>
<td></td>
<td>/ bin</td>
</tr>
<tr>
<td></td>
<td>/ dir</td>
</tr>
<tr>
<td></td>
<td>h o m e</td>
</tr>
<tr>
<td></td>
<td>T H = /</td>
</tr>
<tr>
<td>0x7fffffff0</td>
<td>r: \0 PA</td>
</tr>
<tr>
<td></td>
<td>e: / home</td>
</tr>
<tr>
<td></td>
<td>/ home</td>
</tr>
<tr>
<td></td>
<td>O M E =</td>
</tr>
<tr>
<td>0x7fffffff0</td>
<td>st: \0 H</td>
</tr>
<tr>
<td></td>
<td>rc: \0 d</td>
</tr>
<tr>
<td></td>
<td>cp: \0 s</td>
</tr>
<tr>
<td></td>
<td>Window save area</td>
</tr>
<tr>
<td></td>
<td>(16 words)</td>
</tr>
</tbody>
</table>

%sp+0, 0x7fffffff58

LOW-LEVEL SYSTEM INFORMATION
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.

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.

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**

<table>
<thead>
<tr>
<th>name:</th>
</tr>
</thead>
<tbody>
<tr>
<td>save %sp, -80, %sp</td>
</tr>
<tr>
<td>1: call 2f</td>
</tr>
<tr>
<td>sethi %hi(<em>GLOBAL_OFFSET_TABLE</em>+{.-1b}), %17</td>
</tr>
<tr>
<td>2: or %17, %lo(<em>GLOBAL_OFFSET_TABLE</em>+{.-1b}), %17</td>
</tr>
<tr>
<td>add %17, %o7, %17</td>
</tr>
</tbody>
</table>

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.

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.

When assembling position-independent code, a symbol’s “value” is the offset into the global offset table, not its virtual address.
The large model assumes no limit on global offset table size; it computes the table offset into one register and combines that with %17 to address the desired entry.

**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.
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

<table>
<thead>
<tr>
<th>C</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>extern void function();</td>
<td>.global</td>
</tr>
<tr>
<td>function();</td>
<td>function</td>
</tr>
<tr>
<td></td>
<td>call</td>
</tr>
<tr>
<td></td>
<td>nop</td>
</tr>
</tbody>
</table>

Indirect function calls use the jmp1 instruction. As explained elsewhere, register %o7 holds the return address; so the compiler generates a jmp1 instruction that follows this convention.

Figure 3-41: Absolute Indirect Function Call

<table>
<thead>
<tr>
<th>C</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>extern void (*ptr)();</td>
<td>.global</td>
</tr>
<tr>
<td>extern void name();</td>
<td>ptr, name</td>
</tr>
<tr>
<td>ptr = name;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sethi</td>
</tr>
<tr>
<td></td>
<td>%hi(name), %o0</td>
</tr>
<tr>
<td></td>
<td>or</td>
</tr>
<tr>
<td></td>
<td>%lo(name), %o0</td>
</tr>
<tr>
<td></td>
<td>sethi</td>
</tr>
<tr>
<td></td>
<td>%hi(ptr), %g1</td>
</tr>
<tr>
<td></td>
<td>st</td>
</tr>
<tr>
<td></td>
<td>%o0, [%g1 + %lo(ptr)]</td>
</tr>
<tr>
<td></td>
<td>sethi</td>
</tr>
<tr>
<td></td>
<td>%hi(ptr), %g1</td>
</tr>
<tr>
<td></td>
<td>ld</td>
</tr>
<tr>
<td></td>
<td>[%g1 + %lo(ptr)], %g1</td>
</tr>
<tr>
<td></td>
<td>jmp1</td>
</tr>
<tr>
<td></td>
<td>%g1, %o7</td>
</tr>
<tr>
<td></td>
<td>nop</td>
</tr>
</tbody>
</table>

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.
Branching

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.
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 %10;
- case label constants begin at zero;
- case labels, default, and the address table use assembly names .Lcasei, .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 %10, 4, %g0
bgu .Ldef
sll %10, 2, %10
sethi %hi(.Ltab), %ol
or %o1, %lo(.Ltab), %ol
ld [%10 + %o1], %10
jmp %10, %g0
nop
.Ltab: .word .Lcase0
.word .Ldef
.word .Lcase2
.word .Lcase3
```
Figure 3-46: Position-Independent switch Code (continued)

```plaintext
case 2: sethi %hi(.Ltab - 1b), %g1
    ... 2:  or %g1, %lo(.Ltab - 1b), %g1
case 3: add %10, %g1, %10
    ... ld [%o7 + %10], %10
default: jmpl %o7 + %10, %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.

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.

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 long jmp [see set jmp (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](image)

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.
ELF Header

Machine Information

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

<table>
<thead>
<tr>
<th>Position</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>e_ident [EI_CLASS]</td>
<td>ELFCLASS32</td>
</tr>
<tr>
<td>e_ident [EI_DATA]</td>
<td>ELFDA2DATA</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>.got</td>
<td>SHT_PROGBITS</td>
<td>SHF_ALLOC + SHF_WRITE</td>
</tr>
<tr>
<td>.plt</td>
<td>SHT_PROGBITS</td>
<td>SHF_ALLOC + SHF_WRITE + SHF_EXECINSTR</td>
</tr>
<tr>
<td>.sdata</td>
<td>SHT_PROGBITS</td>
<td>SHF_ALLOC + SHF_WRITE</td>
</tr>
</tbody>
</table>

.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

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte8</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>half16</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>word32</td>
<td>32</td>
<td>0</td>
</tr>
<tr>
<td>disp30</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>disp22</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>imm22</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>simm13</td>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>3-word PLT entry</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

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.
Relocation

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.

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.

This means the place (section offset or address) of the storage unit being relocated (computed using \( r_{offset} \)).

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.

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, \( V\text{-simm13} \) means the the computed value may not have significant, non-zero bits outside the simm13 field.

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

Page 5-1

## Dynamic Linking

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic Section</td>
<td>5-5</td>
</tr>
<tr>
<td>Global Offset Table</td>
<td>5-5</td>
</tr>
<tr>
<td>Function Addresses</td>
<td>5-6</td>
</tr>
<tr>
<td>Procedure Linkage Table</td>
<td>5-7</td>
</tr>
<tr>
<td>Program Interpreter</td>
<td>5-10</td>
</tr>
</tbody>
</table>
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 ($0 \times 10000$) 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

<table>
<thead>
<tr>
<th>File Offset</th>
<th>File</th>
<th>Virtual Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ELF header</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Program header table</td>
<td></td>
</tr>
<tr>
<td>0x100</td>
<td>Text segment</td>
<td>0x10100</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>0x3beff</td>
</tr>
<tr>
<td>0x2bf00</td>
<td>Data segment</td>
<td>0x4bf00</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>0x50cff</td>
</tr>
<tr>
<td>0x30d00</td>
<td>Other information</td>
<td></td>
</tr>
</tbody>
</table>

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.
Program Loading

- 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

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Contents</th>
<th>Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td><strong>Header padding</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x100 bytes</td>
<td></td>
</tr>
<tr>
<td>0x1010</td>
<td><strong>Text segment</strong></td>
<td>Text</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x2be00 bytes</td>
<td></td>
</tr>
<tr>
<td>0x3bf00</td>
<td><strong>Data padding</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x100 bytes</td>
<td></td>
</tr>
<tr>
<td>0x4b000</td>
<td><strong>Text padding</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xf00 bytes</td>
<td></td>
</tr>
<tr>
<td>0x4bf00</td>
<td><strong>Data segment</strong></td>
<td>Data</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x4e00 bytes</td>
<td></td>
</tr>
<tr>
<td>0x50d00</td>
<td>Uninitialized data</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x1024 zero bytes</td>
<td></td>
</tr>
<tr>
<td>0x51d24</td>
<td><strong>Page padding</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x2dc zero bytes</td>
<td></td>
</tr>
</tbody>
</table>

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, illustrat-
ing constant relative positioning. The table also illustrates the base address computations.

**Figure 5-4: Example Shared Object Segment Addresses**

<table>
<thead>
<tr>
<th>Source</th>
<th>Text</th>
<th>Data</th>
<th>Base Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>0x200</td>
<td>0x2a400</td>
<td>0x0</td>
</tr>
<tr>
<td>Process 1</td>
<td>0xc0000200</td>
<td>0xc002a400</td>
<td>0xc0000000</td>
</tr>
<tr>
<td>Process 2</td>
<td>0xc0010200</td>
<td>0xc003a400</td>
<td>0xc0010000</td>
</tr>
<tr>
<td>Process 3</td>
<td>0xd0020200</td>
<td>0xd004a400</td>
<td>0xd0020000</td>
</tr>
<tr>
<td>Process 4</td>
<td>0xd0030200</td>
<td>0xd005a400</td>
<td>0xd0030000</td>
</tr>
</tbody>
</table>
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_GLOBAL_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

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

<table>
<thead>
<tr>
<th>Object File</th>
<th>Memory Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>.PLT0:</td>
<td>.PLT0:</td>
</tr>
<tr>
<td>unimp</td>
<td>save %sp,-64,%sp</td>
</tr>
<tr>
<td>unimp</td>
<td>call dynamic-linker</td>
</tr>
<tr>
<td>unimp</td>
<td>nop</td>
</tr>
<tr>
<td>.PLT1:</td>
<td>.PLT1:</td>
</tr>
<tr>
<td>unimp</td>
<td>.word identification</td>
</tr>
<tr>
<td>unimp</td>
<td>unimp</td>
</tr>
<tr>
<td>unimp</td>
<td>unimp</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>.PLT101:</td>
<td>.PLT101:</td>
</tr>
<tr>
<td>sethi (.-.PLT0),%g1</td>
<td>sethi (.-.PLT0),%g1</td>
</tr>
<tr>
<td>ba,a .PLT0</td>
<td>sethi %hi(name1),%g1</td>
</tr>
<tr>
<td>nop</td>
<td>jmp1 %g1+%lo(name1),%g0</td>
</tr>
<tr>
<td>.PLT102:</td>
<td>.PLT102:</td>
</tr>
<tr>
<td>sethi (.-.PLT0),%g1</td>
<td>sethi (.-.PLT0),%g1</td>
</tr>
<tr>
<td>ba,a .PLT0</td>
<td>sethi %hi(name2),%g1</td>
</tr>
<tr>
<td>nop</td>
<td>jmp1 %g1+%lo(name2),%g0</td>
</tr>
<tr>
<td></td>
<td>nop</td>
</tr>
</tbody>
</table>

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 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 `jmp1` instruction, and then patching the `ba, a` to be `sethi`. If a re-entrant function call occurs between the two word updates, the `jmp1` 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 `jmp1` 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 `jmp1`. 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.
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
System Library

Support Routines

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

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_Q_add</td>
<td>Computes the sum of two quad-precision values.</td>
</tr>
<tr>
<td>_Q_cmp</td>
<td>Compares two quad-precision values and returns a value indicating their relative ordering.</td>
</tr>
<tr>
<td>_Q_cmpq</td>
<td>Compares two quad-precision values and returns a value indicating their relative ordering, using the same values as _Q_cmp.</td>
</tr>
<tr>
<td>_Q_div</td>
<td>Divides one quad-precision value by another.</td>
</tr>
<tr>
<td>_Q dtoq</td>
<td>Converts a quad-precision value to double-precision.</td>
</tr>
<tr>
<td>_Q_ite</td>
<td>Performs a test for equality.</td>
</tr>
<tr>
<td>_Q_neg</td>
<td>Negates a quad-precision value.</td>
</tr>
<tr>
<td>_Q_qtos</td>
<td>Converts a quad-precision value to single-precision.</td>
</tr>
<tr>
<td>_Q_sqrt</td>
<td>Computes the square root of a quad-precision value.</td>
</tr>
<tr>
<td>_Q_stoq</td>
<td>Converts a single-precision value to quad-precision.</td>
</tr>
<tr>
<td>_mul</td>
<td>Multiplies two quad-precision values.</td>
</tr>
<tr>
<td>_rem</td>
<td>Computes the remainder of a division operation.</td>
</tr>
<tr>
<td>_stre1</td>
<td>Performs a test for equality.</td>
</tr>
<tr>
<td>_stre2</td>
<td>Performs a test for equality.</td>
</tr>
<tr>
<td>_stre4</td>
<td>Performs a test for equality.</td>
</tr>
<tr>
<td>_udiv</td>
<td>Divides one quad-precision value by another.</td>
</tr>
<tr>
<td>_umul</td>
<td>Multiplies two quad-precision values.</td>
</tr>
<tr>
<td>_urem</td>
<td>Computes the remainder of a division operation.</td>
</tr>
</tbody>
</table>

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.

```c
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 axexc field of the FSR will be unchanged. Otherwise any exceptions are OR’ed into the axexc 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.

<table>
<thead>
<tr>
<th>Relation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a = b )</td>
<td>0</td>
</tr>
<tr>
<td>( a &lt; b )</td>
<td>1</td>
</tr>
<tr>
<td>( a &gt; b )</td>
<td>2</td>
</tr>
<tr>
<td>( a ) unordered</td>
<td>3</td>
</tr>
</tbody>
</table>

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 axexc field of the FSR will be unchanged. Otherwise any exceptions are OR’ed into the axexc 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 fcmppeq instruction: If any exceptions arise for which the corresponding TEM bits of the FSR are on, a SIGFPE will be generated, and the axexc field of the FSR will be unchanged. Otherwise any exceptions are OR’ed into the axexc 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 \( \frac{a}{b} \) 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 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_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 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_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 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_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 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_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_itow(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_itow 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 fol-
   lows.
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\_u\_to\_q 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\_u\_to\_q is present for the convenience of compilers and has no direct counterpart in the SPARC instruction set.)

\begin{verbatim}
long double \_Q\_sqrt(long double a)
This function corresponds to the SPARC \texttt{f\_sqrt\_q} instruction. It returns the square root of its argument, computed in quad-precision. The following aspects of exception handling mimic the \texttt{f\_sqrt\_q} 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.
\end{verbatim}

\begin{verbatim}
long double \_Q\_stq(float a)
This function corresponds to the SPARC \texttt{f\_stq} 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 \texttt{f\_stq} 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.
\end{verbatim}

\begin{verbatim}
long double \_Q\_qsub(long double a, long double b)
This function corresponds to the SPARC \texttt{f\_sub\_q} instruction. It returns $a - b$ computed in quad-precision. The following aspects of exception handling mimic the \texttt{f\_sub\_q} 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.
\end{verbatim}

\begin{verbatim}
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.)
\end{verbatim}

\begin{verbatim}
int .\_\_di\_v(int a, int b)
This function computes $a / b$ with signed integer division, leaving the result in the caller’s \%0 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 \%01 through \%05 have unspecified values.
\end{verbatim}

\begin{verbatim}
unsigned int \_\_d\_t\_ou(double a)
This function converts the double-precision input argument to an unsigned integer (discarding any fractional part) and returns the unsigned integer value. \_\_d\_t\_ou raises exceptions as follows.
\end{verbatim}
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 \textit{\_\_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 \texttt{SIGFPE} will be generated, and the \textit{aexc} field of the FSR will be unchanged. Otherwise, any exceptions are OR'ed into the \textit{aexc} field of the FSR. (Note that \textit{\_\_dtou} is present for the convenience of compilers and has no direct counterpart in the SPARC instruction set.)

\begin{verbatim}
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 \texttt{SIGFPE} will be generated, and the \textit{aexc} field of the FSR will be unchanged. Otherwise, any exceptions are OR'ed into the \textit{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.)
\end{verbatim}

\begin{verbatim}
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.
\end{verbatim}

\begin{verbatim}
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.
\end{verbatim}

\begin{verbatim}
.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.
\end{verbatim}
%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 %i10 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, stre1, stre2, stre4, and stre8 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 stre2 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 \(a \times b\) 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 \%0 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 \%01 through \%05 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 specified 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>`

```c
extern void __assert(const char *, const char *, int);
#define assert(EX) ((EX) || (__assert(#EX, __FILE__, __LINE__), 0))
```
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 EINVAL 13
#define ENFILE 14
#define ENOTBLK 15
#define EBUSY 16
#define EXIST 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
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 ECURRNG 37
#define EL2SYNC 38
#define EL3HLT 39
#define EL3RST 40
#define ELNRNG 41
#define EUNATCH 42
#define ENOSCI 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 ERRNOET 66
#define ENOLINK 67
#define EADV 68
#define ESPIPE 69
#define ECOMM 70
#define EPROTO 71
#define EMULTIHOP 74
#define EBADMSG 77
#define ENAMEETOOLONG 78
#define EOVERFLOW 79
#define ENOTUNIQ 80
#define EBADF 81
#define ERANGE 82
#define ENOSYS 89
#define ELOOP 90
#define EREREITE 91
#define ESTPIPE 92
#define ENOTEMPTY 93
#define EUSER 94
#define ECONNABORTED 130
#define ECONNRESET 131
#define ECONNREFUSED 146
#define ESTALE 151
```c
#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_NOCOPY 04000
#define O_NOCREATE 0
#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_pid;
  long   l_sysid;
  long   pad[4];
} flock_t;
```

```c
extern int __flt_rounds;
#define FLT_ROUNDS __flt_rounds
```
#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_NOPCK -1
#define MM_OK 0x00
#define MM_NOMSG 0x01
#define MM_NOCON 0x04
Figure 6-10: <ftw.h>

```c
#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>

```c
struct group {
    char *gr_name;
    char *gr_passwd;
    gid_t gr_gid;
    char **gr_mem;
};
```
Figure 6-12: <sys/ipc.h>

```c
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>

```c
#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)

```c
#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
```
System Data Interfaces

Figure 6-14: <limits.h>

```c
#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>

```c
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;
};
```

```c
#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>

```c
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>`

```c
#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>`

```c
#define MS_RDONLY 0x01
#define MS_SSDATA 0x04
#define MS_NOSUID 0x10
#define MS_REMOUNT 0x20
```
Figure 6-19: <sys/msg.h>

```c
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_spad1;
    time_t       msg_rtime;
    long         msg_rpad2;
    time_t       msg_ctime;
    long         msg_rpad3;
    long         msg_rpad4[4];
};
#define MSG_NOERROR  0100000
```
Figure 6-20: `<netconfig.h>`

```
struct netconfig {
    char *nc_netid;
    unsigned long nc_semantics;
    unsigned long nc_flag;
    char *nc_protofmy;
    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_NOPFLAG 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 "nba"
#define NC_ECMAP "ecma"
#define NC_DATAKIT "datakit"
#define NC_CCITT "ccitt"
#define NC_SNA "sna"
#define NC_DECNET "decdnit"
#define NC_DLI "dli"
#define NC_LAT "lat"
#define NC_HYLINK "hylink"
#define NC_APPLETALK "appletalk"
#define NC_NET "net"
#define NC_IEE802 "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_addr;
};

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

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_UNNKN 6
#define ND_NOCCTRL 7
#define ND_FAILCTRL 8
#define ND_SYSTEM 9
#define ND_HOSTSERV 0
#define ND_HOSTSERVLIST 1
#define ND_JSON 2
#define ND_ADDR 3
#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>`**

```c
#define NL_SETD 1
typedef short nl_item;
typedef void *nl_catd;
```

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

```c
#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>`

```c
struct pollfd {
    int fd;
    short events;
    short revents;
};
#define POLLIN   0x0001
#define POLLPRI  0x0002
#define POLLOUT  0x0004
#define POLLRDNORM 0x0040
#define POLLWRNORM POLLOUT
#define POLLRDDBAND 0x0080
#define POLLWRBBAND 0x0100
#define POLLNORM  POLLRDNORM
#define POLLERR   0x0008
#define POLLHUP   0x0010
#define POLLNVAL  0x0020
```
# 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>

```c
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>

```c
#define RLIMIT_CPU     0
#define RLIMITFSIZE    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 RLIMIT_INFINITY 0x7fffffff

typedef unsigned long rlim_t;

struct rlimit {
    rlim_t  rlim_cur;
    rlim_t  rlim_max;
};
```
# define MAX_AUTH_BYTES 400
# define MAXNETNAMELEN 255
# define HEXKEYBYTES 48

e num auth_stat {
    AUTH_OK=0,
    AUTH_BADCRED=1,
    AUTH_REJECTEDCRED=2,
    AUTH_BADVERF=3,
    AUTH_REJECTEDVERF=4,
    AUTH_TOWEAK=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)

```c
struct authsysParms {
    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_CANTDECODEARGS=2,
    RPC_CANTSEND=3,
    RPC_CANTRECV=4,
    RPC_TIMEDOUT=5,
    RPC_INTR=18,
    RPC_UDERROR=23,
    RPC_VERSMISMATCH=6,
    RPC_AUTHERROR=7,
    RPC_PROGINAVAIL=8,
    RPC_PROTOVERSMPATHT=9,
    RPC_PROCVNAVAIL=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)
#define RPC_PMAPFAILURE RPC_RPCBFailure
#define RPC_ANYSOCK -1
#define RPC_ANYFDD 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_REXMIT 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
Figure 6-28: <rpc.h> (continued)

```c
#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;
} SVCXFRPT;

struct svc_req {
    unsigned long rq_prog;
    unsigned long rq_vers;
    unsigned long rq_proc;
    struct opaque_auth rq_cred;
    char *rq_cintcred;
    SVCXFRPT *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)
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;
};
struct call_body {
    unsigned long cb_rpcver;
    unsigned long cb_prog;
    unsigned long cb_ver;
    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_ver;
    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 authdesfullname adc_fullname;
    unsigned long adc_nickname;
};
typedef struct {
    enum xdr_op       x_op;
    struct xdr_ops {
        int (*x_getlong)();
        int (*x_putchar)();
        int (*x_getbytes)();
        int (*x_putchar)();
        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_handly;
} 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_rttaddr)
#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)
```
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>`

```c
#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_nsem;
    time_t sem_ctime;
    long sem_pad1;
    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>`

```c
#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_map;
    unsigned short shm_lkcnt;
    pid_t shm_lpid;
    pid_t shm_cpid;
    unsigned long shm_nattch;
    unsigned long shm_nattch;
    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 SIGFWR 19
#define SIGWINCH 20
#define SIGURG 21
#define SIGFOLL 22
#define SIGSTOP 23
```
System Data Interfaces

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

```c
#define SIGTSTP 24
#define SIGCONT 25
#define SIGTMIN 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(*)(void))-1
#define SIG_IGN (void(*)(void))1
#define SIG_HOLD (void(*)(void))2
#define SIG_DFL (void(*)(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_NOCILDWAIT 0x00001000
#define SA_NOCILDSTOP 0x00002000
```
# 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_FLTSC 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 _sttime;
                } cld;
            }
        }
    }
} (continued on next page)
Figure 6-34: <sys/siginfo.h> (continued)

```c
} _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>

```c
#define _ST_FSTYPSZ 16
struct stat {
    dev_t st_dev;
    long st_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;
    time_t st_atim;
    time_t st_mtim;
    time_t st_ctim;
    long st_blocks;
    long st_blksize;
    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)

```c
#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_IWGRP 00040
#define S_IXGRP 00010
#define S_IRWXO 00007
#define S_IROTH 00004
#define S_IWOTH 00002
#define S_IXOTH 00001

#define S_ISFIFO (mode & S_IFMT) == S_IFIFO
#define S_ISCHR (mode & S_IFMT) == S_IFCHR
#define S_ISDIR (mode & S_IFMT) == S_IFDIR
#define S_ISBLK (mode & S_IFMT) == S_IFBLK
#define S_ISREG (mode & S_IFMT) == S_IFREG
```
#define FSTYPSZ 16

typedef struct statvfs {
    unsigned long f_bsize;
    unsigned long f_fsize;
    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_ssize;
    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

#define _VA_LIST void *

typdef _VA_LIST va_list;

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

extern void va_end(va_list);
#define va_end(list) (void)0
#define NULL 0
typedef int ptdiff_t;
typedef unsigned int size_t;
typedef long wchar_t;
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 _IERR 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 & _IERR)
#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 RPREDAT 0x004
#define RPRTDIS 0x008
#define RPRTNORM 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)
# 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_FDISNERT (STR | 020)
# define I_SENDFD (STR | 021)
# define I_RECVFD (STR | 016)
# define I_SRDPOP (STR | 023)
# define I_GRDOPT (STR | 024)
# define I_LIST (STR | 025)
# define I_FLINK (STR | 026)
# define I_UPUNLINK (STR | 027)
# define I_FLUSHBAND (STR | 034)
# define I_CKRBAND (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 strioct1 {
    int ic_cmd;
    int ic_timeout;
    int ic_lec;
    char *ic_dp;
};

struct strbuf {
    int maxlen;
} (continued on next page)
Figure 6-41: <stropts.h> (continued)

```c
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 char bi_pri;
    int bi_flag;
};
```


```c
#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 VSWITCH 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 CINTR 0177
#define CQUIT 034
#define CERASE '#'
#define CKILL '0'
#define CEOT 04
#define CEOL 0
#define CEOL2 0
#define CEOF 04
#define CSTART 021
#define CSTOP 023
#define CSWITCH 032
#define CNSWITCH 0
#define CSUSP CTRL('z')
#define CDSUSP CTRL('y')
#define CRPRNT CTRL('z')
#define CFLOW CTRL('o')
#define CWERASE CTRL('w')
#define CLNEXT CTRL('v')
#define IGNBRK 0000001
#define BRKINT 0000002
```

(continued on next page)
# define IGNPAR 000004
# define PARMRK 000010
# define INPCK 000020
# define ISTRIP 000040
# define INLCR 000100
# define IGNCR 000200
# define ICRNL 000400
# define IUCLC 001000
# define IXON 002000
# define IXANY 004000
# define IXOFF 010000
# define IMAXBEL 020000
# define OPOST 000001
# define OLCUC 000002
# define ONLCR 000004
# define OCRNL 000010
# define ONOCR 000020
# define ONLRET 000040
# define OFILL 000100
# define OFDEL 000200
# define NLDLY 000400
# define NL0 0
# define NL1 000400
# define CRDLY 000300
# define CR0 0
# define CR1 001000
# define CR2 002000
# define CR3 003000
# define TABDLY 0014000
# define TAB0 0
# define TAB1 000400
# define TAB2 001000
# define TAB3 0014000
# define XTABS TAB3
# define BSDLY 002000
# define BS0 0
# define BS1 002000
# define VTDLY 004000
# define VT0 0
# define VT1 004000
# define FF DLY 0100000
# define FF0 0
# define FF1 0100000
# define CBAUD 000017
# define B0 0
# define B50 000001
# define B75 000002
# define B110 000003
# define B134 000004
# define B150 000005
# define B200 000006
# define B300 000007
# define B600 000010

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

```c
#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 ECHOK 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 TCIFLUSH1 1
#define TCIOFLUSH 2
#define TCOOFF 0
#define TCOON 1
#define TCIOFF 2
#define TCION 3

struct termios {
    tcflag_t c_iflag;
};
```

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

```c

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>`

```c

#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>`

```c

#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>`

```c
#define TCOO_NOPEER 1
#define TCOO_PEERNOROOMQ 2
#define TCOO_PEERBADSTATE 3
#define TCOO_PEERINITIATED 4
#define TCOO_PROVIDERINITIATED 5
#define TCOO_DEFAULTADRSZ 4
```

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

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

type define clock_t;
type define 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];

type define struct timespec {
    time_t tv_sec;
    long tv_nsec;
} timespec_t;

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

struct tms {
    clock_t tms_utime;
    clock_t tms_stime;
    clock_t tms_cutime;
    clock_t tms_cstime;
};
Figure 6-49: `<sys/timod.h>`

```c
#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

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

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

```c
#define T_DATAFER 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_LISTEN 11
#define T_OPEN 0
#define T_OPTMGMT 2
#define T_PASSCONN 24
#define T_RCV 16
#define T_RCVCONECT10
#define T_RCVDIS1 19
#define T_RCVDIS2 20
#define T_RCVDIS3 21
#define T_RCVREL 23
#define T_RCVDATA 6
#define T_RCVDATA 7
#define T_snd 15
#define T_sndDIS1 17
#define T_sndDIS2 18
#define T_sndREL 22
#define T_sndDATA 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 TNOCDATA 13
#define TNODIS 14
#define TNOREL 17
#define TNOTSUPPORT 18
#define TNOUDERR 15
#define TOUTSTATE 6
#define TSTATECHNG 19
#define TSYSSERR 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)

```c
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 unidadata {
    struct netbuf    addr;
    struct netbuf    opt;
    struct netbuf    udata;
};
```

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

```c
#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**

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

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

```c
#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**

```c
#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
```
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 unsigned int size_t;
typedef unsigned long uid_t;
typedef unsigned int size_t;
typedef long off_t;

typedef int gregset_t[19];

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

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

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 fregset_t;

typedef struct {
    gregset_t gregs;
"} (continued on next page)
Figure 6-60: `<ucontext.h>` (continued)

```c
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>`

```c
typedef struct iovec {
    char *iov_base;
    int iov_len;
} iovec_t;
```
Figure 6-62: `<ulimit.h>`

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


```c
#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_CONTROL 1
#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_VERSION 12
#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 _SC_CHOWN_RESTRICTED 9
#define STDIN_FILENO  0
#define STDOUT_FILENO 1
#define STDERR_FILENO 2
```
**Figure 6-64:** `<utime.h>`

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

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

```c
#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>

```c
#define WEXITED 0001
#define WTRAPPED 0002
#define WSTOPPED 0004
#define WCONTINUED 0010
#define UNTRACED WSTOPPED
#define WNORTH 0100
#define WNOWAIT 0200

#define WSTOPFLAG 0177
#define WCOREFLAGS 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) == WSTOPFLAG && WHIBYTE(stat) != 0)
#define WIFCONTINUED(stat) (WWORD(stat) == WCOREFLAG)
#define WEXITSTATUS(stat) WHIBYTE(stat)
#define WTERMSIG(stat) (WLOBYTE(stat) & WSIGMASK)
#define WSTOPSIG(stat) WHIBYTE(stat)
#define WCOREDUMP(stat) ((stat) & WCOREFLAG)
```
X Window Data Definitions

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>

```c
extern WidgetClass compositeWidgetClass;
```

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

```c
extern WidgetClass constraintWidgetClass;
```

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

```c
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

```c
#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
```
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       *XtApplicationContext;
typedef unsigned long XtValueMask;
typedef unsigned long XtIntervalId;
typedef unsigned long XtInputId;
typedef unsigned long XtWorkProcId;
typedef unsigned int  XtGeometryMask;
typedef unsigned long  XtGMask;
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;
typedef void *XtTranslations;
typedef void *XtAccelerators;
typedef unsigned int Modifiers;

#define XtCWQueryOnly (1 << 7)
#define XtSMDon'tChange 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;
#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;
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[];
# define XtAllEvents ((EventMask) -1L)
# define XtIMEvent 1
# define XtITimer 2
# define XtIMAAlternateInput 4
# define XtIMAAll (XtIMEvent | XtITimer | XtIMAAlternateInput)

# 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>`

```c
extern WidgetClass vendorShellWidgetClass;
```

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

```c
typedef unsigned long XID;
typedef XID Window;
typedef XID Drawable;
typedef XID Font;
typedef XIDPixmap;
typedef XID Cursor;
typedef XIDColormap;
typedef XIDGCContext;
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

```c
#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)
```
# 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 LASTEEvent 35 /* must be bigger than any event */
# 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 NotifyNonlinear  3
# define NotifyNonlinearVirtual  4
# define NotifyPointer  5
# define NotifyPointerRoot  6
# define NotifyDetailNone  7
/*
 * Define visibility:
 *
 * #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: &lt;X11/X.h&gt;, Part 6 of 12

```c
#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)
```
# define CNX (1<<0)
# define CNY (1<<1)
# define CNWidth (1<<2)
# define CNHeight (1<<3)
# define CNBorderWidth (1<<4)
# define CNSibling (1<<5)
# define CNStackMode (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
#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 GXxorReverse 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
```c
#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 YXBand 3

#define CoordModeOrigin 0
#define CoordModePrevious 1

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

#define ArcChord 0
#define ArcPieSlice 1
```
```c
#define GCFunctorion (1L<<0)
#define GCPlaneMask (1L<<1)
#define GCForeground (1L<<2)
#define GCBbackground (1L<<3)
#define GLineWidth (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 GCClipOrigin (1L<<17)
#define GCClipOrigin (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

```c
#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

```c
#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

```c
#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

```c
#define XcmsFailure 0
#define XcmsSuccess 1
#define XcmsSuccessWithCompression 2
#define XcmsUndefinedFormat (XcmsColorFormat)0x00000000
#define XcmsCIELuvYFormat (XcmsColorFormat)0x00000001
#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

```c
typedef struct {
    XcmsFloat red;
    XcmsFloat green;
    XcmsFloat blue;
} XcmsRGBi;

typedef struct {
    XcmsFloat X;
    XcmsFloat Y;
    XcmsFloat Z;
} XcmsCIELab;
```
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;
 typedef struct {
    union {
        XcmsRGB       RGB;
        XcmsRGBi      RGBi;
        XcmsCIEXYZ    CIEXYZ;
        XcmsCIELuvY   CIELuvY;
        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];
 } XcmsPerScrInfo;

typedef void *XcmsCCC;

typedef Status (*XcmsConversionProc)();

typedef XcmsConversionProc *XcmsFuncListPtr;
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;
```c
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)
```

```c
typedef void XExtData;

typedef void XExtCodes;

typedef struct {
    int depth;
    int bits_per_pixel;
    int scanline_pad;
} XPixmapFormatValues;
```
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 ta_x_origin;
    int ta_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;
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;
typedef struct {
    XExtData *ext_data;
    int depth;
    int bits_per_pixel;
    int scanline_pad;
} ScreenFormat;

define 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;
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;
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;
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;
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;
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;
    unsigned int state;
    int mode;
    int detail;
    Bool same_screen;
    Bool focus;
} 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];
} XKeyEvent;

typedef struct {
    int type;
    unsigned long serial;
    Bool send_event;
    Display *display;
    Window window;
    int x, y;
    int width, height;
    int count;
} XExposeEvent;
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;
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;
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;
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;
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;
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;
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;
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;
typedef union __XEvent {
    int type;
    XAnyEvent xany;
    XKeyEvent xkey;
    XButtonDownEvent 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;
#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;
    unsigned direction;
    unsigned min_char_or_byte2;
    unsigned max_char_or_byte2;
    unsigned min_byte;
    unsigned max_byte;
    Bool all_chars_exist;
    unsigned default_char;
    int n_properties;
    XFontProp *properties;
    XCharStruct min_bounds;
    XCharStruct max_bounds;
    XCharStruct *per_char;
    int ascent;
    int descent;
} XFontStruct;
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;
} XDataObject;

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;
} XmTextItem;

typedef struct {
    wchar_t *chars;
    int      nchars;
    int      delta;
    XFontSetfont_set;
} XwTextItem;

typedef void (*XIMProc)();

typedef void *XIM;
typedef void *XIC;

typedef unsigned long XIMStyle;

typedef struct {
    unsigned short count_styles;
    XIMStyle *supported_styles;
} XIMStyles;

#define XIMFreeditArea       0x0001L
#define XIMFreeditCallbacks  0x0002L
#define XIMFreeditPosition   0x0004L
#define XIMFreeditNothing    0x0008L
#define XIMFreeditNone       0x0010L
#define XIMStatusArea       0x0100L
#define XIMStatusCallbacks  0x0200L
#define XIMStatusNothing    0x0400L
#define XIMStatusNone       0x0800L
```c
#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 XNBackgroundColor "backgroundPixmap"
#define XNFontSet "fontSet"
#define XNLineSpace "lineSpace"
#define XNCursor "cursor"
```
#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;
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;

typedef enum {
    XIMTextType, XIMBitmapType
} XIMStatusDataType;

typedef struct _XIMStatusDrawCallbackStruct {
    XIMStatusDataType type;
    union {
        XIMText *text;
        Pixmap bitmap;
    } data;
} XIMStatusDrawCallbackStruct;
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;
# 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)
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
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;
#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_switc))
        | (((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;
# 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 XCMEM 1
# define XCNODENT 2
TCP/IP Data Definitions

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.
System Data Interfaces

Figure 6-137: <netinet/in.h>

```c
#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>

```c
#define IPOPT_EOL  0
#define IPOPT_NOP  1
#define IPOPT_LSRR 131
#define IPOPT_SSRR 137
```

Figure 6-139: <netinet/tcp.h>

```c
#define TCP_NODELAY  0x01
```
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:

\[
\text{as} \quad \text{cc} \quad \text{ld} \\
\text{m4} \quad \text{lex} \quad \text{yacc}
\]

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.
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 or 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/sxXX
/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.
Index

A

ABDAY_ constants 6: 20
ABI Compliance 7: 1
ABI conformance 1: 2, 2: 1, 3: 1, 25–28
see also undefined behavior 3: 1
see also unspecified property 3: 1
ABI conforming application 6: 10–11
ABI conforming system 6: 9, 11
ABI Development Environment 7: 2
ABMON_ constants 6: 20
absolute code 3: 34, 5: 3
see also position-independent code 3: 34
accepted_reply 6: 36
accept_stat 6: 35
ACTION 6: 38
address
  absolute 3: 47
  stack 3: 30
  stack object 3: 47
  unaligned 3: 2, 25–26, 45
  virtual 5: 1
addressing, virtual (see virtual addressing)
aexec FSR field 6: 5
aggregate 3: 3
alignment
  array 3: 3
  bit-field 3: 5
  double and long double 3: 2
  executable file 5: 1
  floating-point arguments 3: 45
  scalar types 3: 1
  stack frame 3: 10, 45–46
  structure and union 3: 3
  trapping 3: 25
allocation, dynamic stack space 3: 44–45
ancillary state registers 3: 28
ANSI, C (see C language, ANSI)
Application Environment 8: 1
architecture
  implementation 3: 1
  processor 3: 1
archive file 7: 2
argv 3: 27
ARG_MAX 6: 21
arguments
  bad assumptions 3: 44
  exec(BA_OS) 3: 27
  floating-point 3: 15–16
  function 3: 8
  incoming 3: 13
  integer 3: 15
  main 3: 27
  outgoing 3: 13
  pointer 3: 15
  quad-precision 3: 16
  register 3: 10
  sign extension 3: 15
  stack 3: 10, 13, 15
  structure and union 3: 16
  unaligned (see address, unaligned)
  variable list 3: 44
argv 3: 27
array 3: 3
as 7: 1
assert (EX) 6: 11
assert.h 6: 11
atexit (BA_OS). 3: 30
AUTH 6: 32
AUTH_ constants 6: 31–33
authdes_cred 6: 37
authdes_fullname 6: 37
authdes_namekind 6: 37
auth_destroy 6: 38
auth_stat 6: 31
authsys_parms 6: 32
automatic variables 3: 44
auxiliary vector 3: 30

B

ba, a instruction, dynamic linking 5: 8
bandinfo 6: 51
base address 3: 32, 4: 4, 5: 3
behavior, undefined (see undefined behavior)
bit-field 3: 5
alignment 3: 5
allocation 3: 5
unnamed 3: 5
boot parameters (see tunable parameters)
branch instructions 3: 42
breakpoints 3: 25
BUFSIZ 6: 47
BUS_ constants 6: 42
Index

C

C language
  ANSI  3: 1, 27, 44, 6: 10–11
  calling sequence  3: 8, 44–45
  fundamental types  3: 1
  main  3: 27
  portability  3: 44
  switch statements  3: 42
  call by value  3: 16
  call instruction  3: 9, 36, 39, 6: 8
  calling sequence  3: 8
  function epilogue  3: 11, 17, 19
  function prologue  3: 11
  function prologue and epilogue  3: 36
  CANBIZ  6: 27
  cartridge tape  2: 1
  char  3: 2
  CHILD_MAX  6: 21
  CLD_constants  6: 42
  clearerr  6: 47
  CLGET_constants  6: 34
  CLIENT  6: 34
  clnt_call  6: 38
  clnt_control  6: 38
  clnt_destroy  6: 38
  clnt_freeers  6: 38
  clnt_geterr  6: 38
  clnt_stat  6: 33
  CLSET__constants  6: 34
  code generation  3: 34
  code sequences  3: 34
  Composite.h  6: 70
  condition codes
    floating-point  3: 28, 6: 2–4
    integer  3: 25, 27–28, 6: 6–9
  configuration parameters (see tunable parameters)
  Constraint.h  6: 70
  coprocessor  3: 28
  Core.h  6: 70
  cp_disabled trap  3: 24
  cp_exception trap  3: 24
  cttype.h  6: 12
  cursorfont.h  6: 73

D

data
  process  3: 21
  uninitialized  5: 2
  data representation  3: 1
  data_access_error trap  3: 24
  data_access_exception trap  3: 24
  data_store_error trap  3: 24
  DAY__constants  6: 20
  daylight  6: 57
  delay instruction  3: 13, 17, 19
  des_block  6: 31
  Development Environment  7: 1–2
  DIR  6: 13
  dirent  6: 13
  dirent.h  6: 13
  .div  6: 1, 6
  division by zero  3: 25, 6: 6–7, 9
  division_by_zero trap  3: 24
  div_t  6: 48
  double  3: 2
  doubleword  3: 1–2, 10, 45
  __dtou  6: 1, 6
  dynamic frame size  3: 44
  dynamic linking  3: 21, 5: 5
    environment  5: 10
    lazy binding  5: 10
    LD_BIND_NOW  5: 10
    re-entrancy  5: 9
    register usage  5: 8
    relocation  5: 8–9
    see also dynamic linker  5: 5
    dynamic segments  3: 22, 5: 3
    dynamic stack allocation  3: 45
      signals  3: 46

E

ELF  4: 1
  emulation, instructions  3: 1
    _end  6: 10
  ENTRY  6: 38
  environment  5: 10
    exec(BA_OS)  3: 27
    envp  3: 27
    errno  6: 14
    errno.h  6: 14
  exceptions (see traps)
exec(BA_OS)  3: 29, 35
interpreter  3: 31
paging  5: 1
process initialization  3: 27
executable file, segments  5: 3
execution mode (see processor execution mode)

F
F_ constants  6: 16, 66
=percent=f0, floating-point return value  3: 17
faddq  6: 1
faults (see traps)
fcc FSR field (see condition codes, floating-point)
FCHR_MAX  6: 21
fcmpleq instruction  6: 2-3
fcmpq  6: 2
fcmpq instruction  6: 3-4
fcnt1.h  6: 16
FD_ constants  6: 16
fdivq instruction  6: 2
fd_set  6: 35
fdtoiq instruction  6: 2
FEEDBACK_ constants  6: 34
feof  6: 47
ferror  6: 47
FILE  6: 48
file, object (see object file)
file offset  5: 1
FILENAME_MAX  6: 47
fileno  6: 47
fitoq instruction  6: 4
float  3: 2
float.h  6: 17
floating-point
  arguments  3: 15-16
  return value  3: 13, 17-18, 6: 7
  state register  3: 14
floating-point state register  6: 6
  initial value  3: 27
flock  6: 16
flock_t  6: 16
floppy disk  2: 1
FLT_rounds  6: 17
  _flt_rounds  6: 17
fntmsg.h  6: 17
fmulq instruction  6: 4
fnegs instruction  6: 4
FOPEN_MAX  6: 47
formats
  array  3: 3
  instruction  4: 3
  structure  3: 3
  union  3: 3
FORTRAN
  COMMON  3: 2
  EQUIVALENCE  3: 2
  =percent=fp (see frame pointer)
  =percent=fp  6: 8
  fp_disabled trap  3: 24
  FPE_ constants  6: 42
  fp_exception trap  3: 24
  fpq  6: 64
  fpregset_t  6: 64
  fp_status  6: 64
  fpu  6: 64
  fq  6: 64
  fqtod instruction  6: 4
  fqtod instruction  6: 4
  fqtos instruction  6: 5
frame pointer  3: 13, 37
  initial value  3: 30
frame size, dynamic  3: 45
fsqrtq instruction  6: 5
  =percent=fsr (see floating-point state register)
FSR (see floating-point state register)
  fstoq instruction  6: 5
  fsubq instruction  6: 6
  __ftou  6: 1, 6
FTW  6: 18
FTW_ constants  6: 18
ftw.h  6: 18
function
  address  5: 6
  void  3: 17
function arguments (see arguments)
function call, code  3: 39
function linkage (see calling sequence)
function prologue and epilogue (see calling sequence)

G
percent=gl
  get condition codes  3: 27
  set condition codes  3: 27
Index


dynamic linking 5:9
global offset table 3:35, 4:2, 4-6, 5:5
_GlobalOFFSET_TABLE_  3:36
=percent=17 3:36
relocation 3:35
_GlobalOFFSET_TABLE_ (see global offset table)
gregset_t 6:64
group 6:18
grp.h 6:18

H
halfword 3:1
header files 6:11
heap, dynamic stack 3:45
HOST_constants 6:26
_huge_val 6:10
HUGE_VAL 6:22
_huge_val 6:22
_h_val 6:22
HZ 6:27

I
I_constants 6:50
=percent=10 3:13
=percent=10
see also return value 3:13
integer return value 3:17
=percent=10 6:8
=percent=17 3:13
=percent=17, see also return address 3:13
=percent=17 6:8
idop 6:29
idop_t 6:29
idtype 6:29
idtype_t 6:29
ILL_constants 6:42
illegal_instruction trap 3:19, 24
in registers 3:8, 13
incoming arguments 3:13
in.h 6:135
initialization, process 3:27
instruction formats 4:3
instruction_access_exception trap 3:24
instructions
  coprocessor 3:28
  emulation 3:1
int 3:2
integer arguments 3:15
Intrinsic.h 6:79
_iob 6:48
_IOEOF 6:47
_IOERR 6:47
_IOFBF 6:47
_IOLBF 6:47
_ERRBF 6:47
iovec 6:65
iovec_t 6:65
IPC_constants 6:19
ipc.h 6:19
ipc_perm 6:19, 24
ip.h 6:135
isalnum 6:12
isalpha 6:12
iscntrl 6:12
isdigit 6:12
isgraph 6:12
islower 6:12
isprint 6:12
ispunct 6:12
isspace 6:12
isupper 6:12
isxdigit 6:12

J
_JBLEN 6:39
jmp_buf 6:39
jmpl instruction 3:17, 19, 40
dynamic linking 5:8

L
langinfo.h 6:20
lazy binding 5:10
LC_constants 6:22
lconv 6:22
L_cuserid 6:48
L_cusermid 6:48
ld 7:1
LD_BIND_NOW 5:10
ldarg 6:48
ld(SD_CMD) (see link editor)
lex 7:1
libm 7:2
_lib_version 6:10
limits.h 6:21
link editor 4:6, 5:5, 7
linkage, function (see calling sequence)
LINK_MAX 6:21
local registers 3:8, 13
local variables 3:44
locale.h 6:22
long 3:2
long double 3:2
longjmp(BA_LIB) (see setjmp(BA_LIB))
L_tmpnam 6:48

M
m4 7:1
macro definitions 6:11
main
  arguments 3:27
  declaration 3:27
malloc(BA_OS) 3:23
MAP_constants 6:23
math.h 6:22
MAX_CANON 6:21
MAX_INPUT 6:21
MAXNAMES 6:27
MAXPATHLEN 6:27
MAXSYMLINKS 6:27
MB_LEN_MAX 6:21
mcontext_t 6:65
mem_address_not_aligned_trap 3:24
memory allocation, stack 3:44-45
memory management 3:21
memory_address_not_aligned trap 3:26
MMU_constants 6:17
man.h 6:23
mmap(KES) 3:22-23
MON_constants 6:20
mount.h 6:23
MS_constants 6:23
msg 6:24
MSG_constants 6:49
msg.h 6:24
msg_type 6:35
msgqid_ds 6:24
很快就 6:1, 7

N
NADDR 6:27
NAME_MAX 6:21
NB_SCTR 6:27
NC_constants 6:25
ND_constants 6:26
nd_addrlist 6:26
nd_hostserv 6:26
nd_hostservlist 6:26
netbuf 6:26, 61
netconfig 6:25
netconfig.h 6:25
netdir.h 6:26
_NFILE 6:47
NGROUPS_MAX 6:21
NGROUPS_MAX 6:27
NL_constants 6:21
nl_catd 6:27
nl_item 6:27
nl_types.h 6:27
nop instruction, dynamic linking 5:7-8
null pointer 3:2, 22, 27
dereferencing 3:22
NZERO 6:21

O
O_constants 6:16
=percent=0 3:13
=percent=0 see also return value 3:13
integer return value 3:17
=percent=0 6:7
=percent=0 6:8
=percent=0 3:13
=percent=0 see also return address 3:13
object file 4:1
ELF header 4:1
executable 3:35
executable file 3:35
relocation 4:3
section 4:2
see also archive file 4:1
see also dynamic linking 5:5
see also executable file 4:1
see also relocatable file 4:1
see also shared object file 4:1

Index
Index

segment 5: 1
shared object file 3: 35
special sections 4: 2
symbol table 4: 2
Object.h 6: 79
offset table, global (see global offset table)
opaque_auth 6: 32
OPEN_MAX 6: 21
optimization 3: 12
out registers 3: 8, 13
outgoing arguments 3: 13
overflow
integer 3: 25–26
window 3: 10

P

P__ constants 6: 29
padding, structure and union 3: 3
page size 3: 21, 32, 5: 1
paging 3: 21, 5: 1
performance 5: 1
parameters
function (see arguments)
system configuration (see tunable parameters)
param.h 6: 27
PASS_MAX 6: 21
passwd 6: 30
PATH variable 7: 1
PATH_MAX 6: 21
_PC__ constants 6: 67
PC-relative 3: 35, 42
performance 3: 1
paging 5: 1
physical addressing 3: 21
physical distribution media 2: 1
PIPE_BUF 6: 21
PIPE_MAX 6: 27
pkgmk 7: 2
pkgproto 7: 2
pkgtrans 7: 2
pointer 3: 2
function argument 3: 15
null 3: 2, 22, 27
POLL__ constants 6: 42
pollfd 6: 28
poll.h 6: 28
POP__ constants 6: 29

portability
C_program 3: 44
data alignment 3: 26
instructions 3: 1
position-independent code 3: 34, 36, 5: 3
see also absolute code 3: 34
see also global offset table 3: 35
_POSIX__ constants 6: 21, 66
privileged_instruction trap 3: 24
procedure linkage table 4: 2, 4, 6, 5: 5, 7
procedures (see functions)
process
dead 3: 46
entry point 3: 27
initialization 3: 27
segment 3: 21
size 3: 21
stack 3: 29
virtual addressing 3: 21
processor architecture 3: 1
processor execution mode 3: 24, 27
processor state register 3: 14
initial value 3: 27
processor-specific information 3: 1, 8, 21, 34, 5: 1,
5–7, 6: 1, 7: 1
procset 6: 29
procset.h 6: 29
procset_t 6: 29
program counter, relative addressing (see PC-relative)
program loading 3: 31, 5: 1
PROT__ constants 6: 23
 perennial=psr (see processor state register)
PSR (see processor state register)
P_tmpdir 6: 48
pwd.h 6: 30

Q

_Q__add 6: 1
_Q__cmp 6: 1
_Q__cmpeq 6: 1–2
_Q__div 6: 1–2
_Q__fdiv 6: 1–2
_Q__fge 6: 1, 3
_Q__fgeq 6: 1, 3
_Q__fle 6: 1, 3
Index

Q_flt 6:1,3
Q_fne 6:1,4
QIC-150 tape data format 2:1
Q_itod 6:1,4
Q_mul 6:1,4
Q_neg 6:1,4
Q_qtod 6:1,4
Q_qtoi 6:1,4
Q_qtos 6:1,5
Q_qtoq 6:1,5
Q_sqrt 6:1,5
Q_stoq 6:1,5
Q_sub 6:1,6
quad-precision
  arguments 3:16
  return value 3:18, 6:7
Q untoq 6:1,6

Range checking 3:25
rdasr instruction 3:28
rdy instruction 3:28
RectObj.h 6:79
re-entrancy 3:18, 5:9
registers
  ancillary state registers 3:28
  calling sequence 3:13
  description 3:8, 13
  floating-point 3:8, 12, 14
  flushing 3:25
  global 3:8, 12, 14
  initial values 3:26, 29
  initializing 3:25
  local 3:13
  process initialization 3:27
  processor state register 3:27
  saving 3:14
  scratch 3:14
  signals 3:14
  window 3:8, 11–12
  Y register 3:14, 28
rejected_reply 6:36
reject_stat 6:36
relocation
  global offset table 3:35
  see object file 4:3

quad-precision
  arguments 3:16
  return value 3:18, 6:7
quad-precision
  arguments 3:16
  return value 3:18, 6:7

R
range checking 3:25
rdasr instruction 3:28
rdy instruction 3:28
RectObj.h 6:79
re-entrancy 3:18, 5:9
registers
  ancillary state registers 3:28
  calling sequence 3:13
  description 3:8, 13
  floating-point 3:8, 12, 14
  flushing 3:25
  global 3:8, 12, 14
  initial values 3:26, 29
  initializing 3:25
  local 3:13
  process initialization 3:27
  processor state register 3:27
  saving 3:14
  scratch 3:14
  signals 3:14
  window 3:8, 11–12
  Y register 3:14, 28
rejected_reply 6:36
reject_stat 6:36
relocation
  global offset table 3:35
  see object file 4:3

quad-precision
  arguments 3:16
  return value 3:18, 6:7
quad-precision
  arguments 3:16
  return value 3:18, 6:7

S
S_c constants 6:44, 49
SA_c constants 6:41
save instruction 3:36
S_c constants 6:67
scalar types 3:1
search.h 6:38
secondary storage 3:21
section, object file 5:1
SEEK_c constants 6:66
segment
  dynamic 3:22
  permissions 5:2
  process 3:21–22, 5:1, 6
segment permissions 3:23
SEG V_c constants 6:42
sem 6:39
sembuf 6:39
sem.h 6:39
semid_ds 6:39
<table>
<thead>
<tr>
<th>Topic</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>sethi instruction</td>
<td>3:36–37</td>
</tr>
<tr>
<td>dynamic linking</td>
<td>5:8</td>
</tr>
<tr>
<td>set jmp(BA_LIB)</td>
<td>3:46</td>
</tr>
<tr>
<td>setjmp.h</td>
<td>6:39</td>
</tr>
<tr>
<td>setrlimit(BA_OS)</td>
<td>3:23</td>
</tr>
<tr>
<td>shared object file</td>
<td>3:35</td>
</tr>
<tr>
<td>segments</td>
<td>3:22, 5:3</td>
</tr>
<tr>
<td>Shell.h</td>
<td>6:60</td>
</tr>
<tr>
<td>SHM_constants</td>
<td>6:40</td>
</tr>
<tr>
<td>shm.h</td>
<td>6:40</td>
</tr>
<tr>
<td>shmid_ds</td>
<td>6:40</td>
</tr>
<tr>
<td>SHN.UNDEF</td>
<td>4:2, 5:7</td>
</tr>
<tr>
<td>short</td>
<td>3:2</td>
</tr>
<tr>
<td>SI_constants</td>
<td>6:43</td>
</tr>
<tr>
<td>SIG_constants</td>
<td>6:41</td>
</tr>
<tr>
<td>sigaltstack</td>
<td>6:41</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>6:1–7</td>
</tr>
<tr>
<td>siginfo.h</td>
<td>6:43</td>
</tr>
<tr>
<td>siginfo.h</td>
<td>6:42</td>
</tr>
<tr>
<td>siginfo_t</td>
<td>6:43</td>
</tr>
<tr>
<td>_SIGJBLLEN</td>
<td>6:39</td>
</tr>
<tr>
<td>sigjmp_buf</td>
<td>6:39</td>
</tr>
<tr>
<td>sign extension</td>
<td></td>
</tr>
<tr>
<td>arguments</td>
<td>3:15</td>
</tr>
<tr>
<td>bit-field</td>
<td>3:5</td>
</tr>
<tr>
<td>signal(BA_OS)</td>
<td>3:14, 24</td>
</tr>
<tr>
<td>signal.h</td>
<td>6:41</td>
</tr>
<tr>
<td>signals</td>
<td>3:14, 46</td>
</tr>
<tr>
<td>signed</td>
<td>3:2, 5</td>
</tr>
<tr>
<td>sigset_t</td>
<td>6:41</td>
</tr>
<tr>
<td>sizeof</td>
<td>3:1–2</td>
</tr>
<tr>
<td>structure</td>
<td>3:3</td>
</tr>
<tr>
<td>=percent=sp (see stack pointer)</td>
<td></td>
</tr>
<tr>
<td>SPARC</td>
<td>3:1, 8, 21, 44, 4:3, 5:1</td>
</tr>
<tr>
<td>SS_constants</td>
<td>6:41</td>
</tr>
<tr>
<td>ST_constants</td>
<td>6:45</td>
</tr>
<tr>
<td>stack</td>
<td></td>
</tr>
<tr>
<td>address</td>
<td>3:30</td>
</tr>
<tr>
<td>arguments</td>
<td>3:13</td>
</tr>
<tr>
<td>dynamic allocation</td>
<td>3:45</td>
</tr>
<tr>
<td>function</td>
<td>3:8</td>
</tr>
<tr>
<td>growth</td>
<td>3:10</td>
</tr>
<tr>
<td>initial process</td>
<td>3:29</td>
</tr>
<tr>
<td>process</td>
<td>3:21–22</td>
</tr>
<tr>
<td>system management</td>
<td>3:23</td>
</tr>
<tr>
<td>stack frame</td>
<td>3:8, 11, 44</td>
</tr>
<tr>
<td>alignment</td>
<td>3:10, 30, 32, 45–46</td>
</tr>
<tr>
<td>organization</td>
<td>3:10, 44</td>
</tr>
<tr>
<td>size</td>
<td>3:11, 44</td>
</tr>
<tr>
<td>stack pointer</td>
<td>3:13, 37</td>
</tr>
<tr>
<td>alignment</td>
<td>3:32</td>
</tr>
<tr>
<td>initial value</td>
<td>3:30</td>
</tr>
<tr>
<td>stat</td>
<td>6:44</td>
</tr>
<tr>
<td>stat.h</td>
<td>6:44</td>
</tr>
<tr>
<td>statvfs</td>
<td>6:45</td>
</tr>
<tr>
<td>statvfs.h</td>
<td>6:45</td>
</tr>
<tr>
<td>statvfs.t</td>
<td>6:45</td>
</tr>
<tr>
<td>&lt;stdio.h&gt;</td>
<td>3:44</td>
</tr>
<tr>
<td>stdargh</td>
<td>6:45</td>
</tr>
<tr>
<td>stddef.h</td>
<td>6:46</td>
</tr>
<tr>
<td>stderr</td>
<td>6:47</td>
</tr>
<tr>
<td>stdin</td>
<td>6:47</td>
</tr>
<tr>
<td>STDIN_constants</td>
<td>6:67</td>
</tr>
<tr>
<td>stdio.h</td>
<td>6:47</td>
</tr>
<tr>
<td>stdlib.h</td>
<td>6:48</td>
</tr>
<tr>
<td>stdout</td>
<td>6:47</td>
</tr>
<tr>
<td>strbuf</td>
<td>6:50</td>
</tr>
<tr>
<td>.stret1</td>
<td>6:1, 7</td>
</tr>
<tr>
<td>.stret2</td>
<td>6:1, 7</td>
</tr>
<tr>
<td>.stret3</td>
<td>6:1, 7</td>
</tr>
<tr>
<td>.stret4</td>
<td>6:1, 7</td>
</tr>
<tr>
<td>.stret8</td>
<td>6:1, 7</td>
</tr>
<tr>
<td>strfdinsert</td>
<td>6:50</td>
</tr>
<tr>
<td>strictl</td>
<td>6:50</td>
</tr>
<tr>
<td>str_list</td>
<td>6:51</td>
</tr>
<tr>
<td>str_mlist</td>
<td>6:51</td>
</tr>
<tr>
<td>stropts.h</td>
<td>6:49</td>
</tr>
<tr>
<td>strpeek</td>
<td>6:50</td>
</tr>
<tr>
<td>strreccvf.d</td>
<td>6:50</td>
</tr>
<tr>
<td>structure</td>
<td>3:3</td>
</tr>
<tr>
<td>function argument</td>
<td>3:16</td>
</tr>
<tr>
<td>padding</td>
<td>3:3</td>
</tr>
<tr>
<td>return value</td>
<td>3:18, 6:7</td>
</tr>
<tr>
<td>STT.FUNC</td>
<td>5:7</td>
</tr>
<tr>
<td>supervisor mode (see processor execution mode)</td>
<td></td>
</tr>
<tr>
<td>svc_destroy</td>
<td>6:38</td>
</tr>
<tr>
<td>svc_fdset</td>
<td>6:35</td>
</tr>
<tr>
<td>svc_freeargs</td>
<td>6:38</td>
</tr>
<tr>
<td>svc_getargs</td>
<td>6:38</td>
</tr>
<tr>
<td>svc_getrproc.caller</td>
<td>6:38</td>
</tr>
<tr>
<td>svc_req</td>
<td>6:35</td>
</tr>
<tr>
<td>SVCXPRT</td>
<td>6:34</td>
</tr>
<tr>
<td>SVID</td>
<td>7:2</td>
</tr>
<tr>
<td>switch statements</td>
<td>3:42</td>
</tr>
<tr>
<td>sysconf(BA_OS)</td>
<td>3:21, 32</td>
</tr>
<tr>
<td>system calls</td>
<td>3:25</td>
</tr>
<tr>
<td>System Headers</td>
<td>7:2</td>
</tr>
<tr>
<td>system load</td>
<td>3:21</td>
</tr>
<tr>
<td>System V ABI</td>
<td>1:1, 6:10</td>
</tr>
</tbody>
</table>
T

T_constants 6: 58–59, 62
tag_overflow trap 3: 24
t_bind 6: 61
t_call 6: 61
tcpip/tcp.h 6: 135
t_discon 6: 61
TEM FSR field 6: 5
termination, process 3: 46
termios 6: 55
termios.h 6: 52
text
  process 3: 21
  sharing 3: 35
ticlts.h 6: 55
ticots.h 6: 55
ticotsord.h 6: 55
tihdr.h 6: 56
time.h 6: 57
times.h 6: 58	imestruc 6: 57	imestruc_t 6: 57	imexval 6: 57	timezone 6: 57
timod.h 6: 58
t_info 6: 61	iuser.h 6: 58
error return values 6: 60
events bitmasks 6: 62
fields of structures 6: 62
flags 6: 63
  service types 6: 58
  structure types 6: 62
  transport interface data structures 6: 61
  transport interface states 6: 59
  user-level events 6: 59
tm 6: 57
TMP_MAX 6: 21
tms 6: 58
toascii 6: 12
  _tolower 6: 12
t_optmgmt 6: 61
  _toupper 6: 12
TRAP_constants 6: 42
trap_instruction trap 3: 24
traps 6: 6–7, 9
  access exception 3: 22
  illegal_instruction 3: 19
  interface 3: 24
  signals 3: 24
  type table 3: 25
t_uderr 6: 61
tunable parameters
  process size 3: 21
  stack size 3: 23
t_unitdata 6: 61
type mismatch 3: 19
types.h 6: 63
tzname 6: 57

U

ucontext 6: 65
ucontext.h 6: 64
ucontext_t 6: 65
.udiv 6: 1, 9
.uio.h 6: 65
UL_constants 6: 65
ulimit.h 6: 65
.ulim 6: 1, 9
unaligned address (see address, unaligned)
undefined behavior 3: 1, 13, 19, 26, 30, 5: 2, 6: 9
  see also ABI conformance 3: 1
  see also unspecified property 3: 1
underflow, window 3: 10
unimp_instruction 3: 19, 6: 8
dynamic linking 5: 8
unimplemented instruction (see traps, illegal_instruction)
uninitialized data 5: 2
union 3: 3
  function argument 3: 16
  return value 3: 18, 6: 7
unistd.h 6: 66
unsigned 3: 2, 5
unspecified property 3: 1, 13–14, 18, 25–26, 28–29,
  5: 1–2, 7–8, 6: 2–9
  see also ABI conformance 3: 1
  see also undefined behavior 3: 1
.urem 6: 1, 9
user mode (see processor execution mode)
utilbuf 6: 67
utime.h 6: 67
utsname 6:67
utsname.h 6:67

V

[varargs.h] 3:44
variable argument list 3:10, 15, 44
variables, automatic 3:44
Vendor.h 6:80
virtual addressing 3:21, 35
  bounds 3:22
  invalid 3:22
VISIT 6:38
void functions 3:17

W

wait.h 6:69
window overflow 3:10
window registers 3:11
  (see registers, window)
  flushing 3:25
  initializing 3:25–26
window underflow 3:10
window_overflow trap 3:24, 26
window_underflow trap 3:24
word 3:1
wrasr instruction 3:28
wry instruction 3:28

X

Xatom.h 6:95
Xcms.h 6:100
XDR 6:37
xdr_destroy 6:38
xdr_discrim 6:37
xdr_getpos 6:38
xdr_inline 6:38
xdr_op 6:37
xdr_setpos 6:38
X.h 6:92
Xlib.h 6:126
_XOPEN_VERSION 6:66
XPRT_constants 6:34
xprt_stat 6:34
Xresource.h 6:128

Xutil.h 6:133

Y

Y register 3:14, 28
yacc 7:1

Z

zero
  division by 3:25, 6:6–7, 9
  null pointer 3:2, 22
  uninitialized data 5:2
  virtual address 3:22
  zero fill 3:5