SYSTEM V
APPLICATION BINARY INTERFACE

MIPS® RISC Processor Supplement

3rd Edition

Document Version: 3
February 1996

Table of Contents

The MIPS Processor and System V ABI

The System V Application Binary Interface (ABI) defines a system interface for compiled application programs. It establishes a standard binary interface for application programs on systems that implement the interfaces defined in the *System V Interface Definition, Third Edition*. This includes systems that have implemented UNIX® System V, Release 4.

This document supplements the generic *System V ABI,* and it contains information specific to System V implementations built on the MIPS® RISC processor architecture. These two documents constitute the complete System V Application Binary Interface specification for systems that implement the MIPS RISC processor architecture.

How to Use the MIPS ABI Supplement

This document contains information referenced in the generic *System V ABI* that may differ when System V is implemented on different processors. Therefore, the generic Application Binary Interface 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 available reference documents, especially *MIPS RISC Architecture* (Copyright © 1990, MIPS Computer Systems, Inc., ISBN 0-13-584749-4). All the information referenced by this supplement is 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 three-year intervals. Each new edition will contain extensions and additions to increase the capabilities of applications that conform to the *ABI*.

As with the *System V Interface Definition*, the *ABI* implements Level 1 and Level 2 support for its constituent parts. Level 1 support indicates a portion of the specification that will be supported indefinitely, while Level 2 support indicates a portion of the specification that may be withdrawn or altered when the next edition of the *System V ABI* is made available.

All components of this document and the generic *System V ABI* have Level 1 support unless they are explicitly labeled as Level 2.

Software Distribution Formats

Physical Distribution Media

The approved media for physical distribution of *ABI*-conforming software are listed below. *ABI*-conforming systems are not required to accept any of these media. A conforming system can install all software through its network connection.

- 60 MByte 1/4-inch cartridge tape in QIC-24 format[1]
- 20 MByte 1/4-inch cartridge tape in QIC-120 format [2]
- 1/2-inch, 9-track magnetic tape recorded at 1600 bpi
- 1.44 MByte 3 1/2-inch floppy disk: double-sided, 80 cylinders/side, 18 sectors/cylinder, 512 bytes/sector
- DDS Recording Format for Digital Audio Tape (DAT) DDS01 Rev E - January, 1990 [3]
- CD-ROM, ISO 9660 with Rockridge extensions

---

1. The QIC-24 cartridge tape data format is described in *Serial Recorded Magnetic Tape Cartridge for Information Interchange (9 tracks, 10,000 FTPI, GCR, 60MB),* Revision D, April 22, 1983. This document is available from the Quarter-Inch Committee (QIC) through Freeman Associates, 311 East Carillo St., Santa Barbara, CA 93101.
2. The QIC-120 cartridge tape data format is described in *Serial Magnetic Tape Cartridge for Information Interchange, Fifteen Track, 0.250 in (6.30mm), 10,000 bpi (394 bpmm) Streaming Mode Group Code Recording,* Revision D, February 12, 1987. This document is available from the Quarter-Inch Committee (QIC) through Freeman Associates, 311 East Carillo St., Santa Barbara, CA 93101
3. The DDS recording format is specified in ANSI Standard X3B5/88-185A, DDS Recording Format.

Machine Interface

Processor Architecture

MIPS RISC Architecture processor (Copyright © 1990, MIPS Computer Systems, Inc., ISBN 0-13-584749-4) defines the processor architecture for two separate Instruction Set Architectures (ISA), MIPS I and MIPS II. The MIPS I Instruction Set Architecture provides the architectural basis for this processor supplement to the generic *ABI*. Programs intended to execute directly on a processor that implements this ISA use the instruction set, instruction encodings, and instruction semantics of the architecture. Extensions available in the MIPS II ISA are explicitly not a part of this specification.

Three points deserve explicit mention.

- A program can assume all documented instructions exist.
- A program can assume all documented instructions work.
- A program can use only the instructions defined by the MIPS I ISA. In other words, *from a program's perspective*, the execution environment provides a complete and working implementation of the MIPS I ISA.

This does not mean 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 MIPS I ISA as a subset, providing additional instructions or capabilities, e.g., the R6000 processor. Programs that use those capabilities explicitly do not conform to the *MIPS ABI*. Executing those programs on machines without the additional capabilities gives undefined behavior.

Data Representation

Byte Ordering

The architecture defines an 8-bit **byte**, 16-bit **halfword**, a 32-bit **word**, and a 64-bit **doubleword**. By convention there is also a 128-bit **quadword**. Byte ordering defines how the bytes that make up halfwords, words, doublewords, and quadwords are ordered in memory. Most significant byte (MSB) byte ordering, or big endian as it is sometimes called, means that the most significant byte is located in the lowest addressed byte position in a storage unit (byte 0).

Although the MIPS processor supports either big endian or little endian byte ordering, an ABI-conforming system must support big endian byte ordering.

The figures below illustrate the conventions for bit and byte numbering within various width storage units. These conventions hold for both integer data and floating-point data, where the most significant byte of a floating-point value holds the sign and at least the start of the exponent.

Figure 3-1: Bit and Byte Numbering in Halfwords

0 msb 15 8	1 lsb 7 0

Figure 3-2: Bit and Byte Numbering in Words

0 msb 31 24	1 23 16	2 15 8	3 lsb 7 0

Figure 3-3: Bit and Byte Numbering in Doublewords

0 msb 31 24	1 23 16	2 15 8	3 7 0
4 31 24	5 23 16	6 15 8	7 lsb 7 0

Figure 3-4: Bit and Byte Numbering in Quadwords

0 msb 31 24	1 23 16	2 15 8	3 7 0
4 31 24	5 23 16	6 15 8	7 7 0
8 31 24	9 23 16	10 15 8	11 7 0
12 31 24	13 23 16	14 15 8	15 lsb 7 0

Fundamental Types

Figure 3-5 shows the correspondence between ANSI C's scalar types and the processor's.

Figure 3-5: Scalar Types

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

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

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

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

Aggregates and Unions

Aggregates (structures and arrays) and unions assume the alignment of their most strictly aligned components. The size of any object, including aggregates and unions, is always a multiple of the alignment of the object. 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.
- If necessary, a structure's size is increased to make it a multiple of the alignment. This may require *tail padding*, depending on the last member.

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

Figure 3-6: Structure Smaller Than a Word

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

Byte aligned, `sizeof` is 1

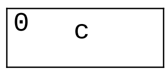


Figure 3-7: No Padding

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

Word aligned, `sizeof` is 8

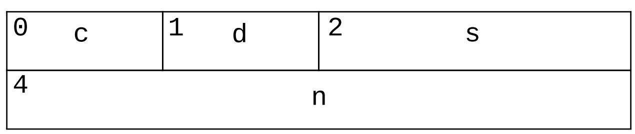


Figure 3-8: Internal Padding

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

Halfword aligned, `sizeof` is 4

0 c	1 *pad*
2 s	

Figure 3-9: Internal and Tail Padding

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

Doubleword aligned, `sizeof` is 24

0 c	1 *pad*
4 *pad*	
8 d	
12 d	
16 s	18 *pad*
20 *pad*	

Figure 3-10: union Allocation

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

Word aligned, `sizeof` is 4

0 c	1 *pad*
0 s	2 *pad*
0 j	

Bit–Fields

C struct and union definitions can have *bit-field*s, defining integral objects with a specified number of bits. Figure 3-11 lists the bit-field ranges.

Figure 3-11: Bit–Field Ranges

Bit-field Type	Width *w*	Range

`signed char`
`char`
`unsigned char`
	1 to 8	-2^{w-1} to $2^{w-1}-1$
		0 to 2^{w-1}
		0 to 2^{w-1}

`signed short`
`short`
`unsigned short`
	1 to 16	-2^{-1} to $2^{w-1}-1$
		-2^{w-1} to $2^{w-1}-1$
		0 to 2^{w-1}

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

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

Plain bit-fields always have signed or unsigned values depending on whether the basic type is signed or unsigned. In particular, `char` bit-fields are unsigned while short, int, and long bit-fields are signed. A signed or unsigned modifier overrides the default type.

In a signed bit-field, the most significant bit is the sign bit; sign bit extension occurs when the bit-field is used in an expression. Unsigned bit-fields are treated as simple unsigned values.

Bit-fields follow 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 reside entirely in a storage unit that is appropriate for its declared type. Thus a bit-field never crosses its unit boundary. However, an unnamed bit-field of non-zero width is allocated in the smallest storage unit sufficient to hold the field, regardless of the defined type.

- Bit-fields can 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 types of bit-fields do not affect the alignment of a structure or union, although member offsets of individual bit-fields follow the alignment constraints.

NOTE The X3J11 ANSI C specification only allows bit–fields of type int, with or without a signed or unsigned modifier.

Figures 3-12 through 3-17 provide examples that show the byte offsets of struct and union members in the upper left corners.

Figure 3-12: Bit Numbering

`0x01020304`

0 01 (31–24)	1 02 (23–16)	2 03 (15–8)	3 04 (7–0)

Figure 3-13: Left-to-Right Allocation

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

Word aligned, sizeof is 4

0 j (31–27)	k (26–21)	m (20–14)	*pad* (13–0)

Figure 3-14: Boundary Alignment

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

Word aligned, sizeof is 12

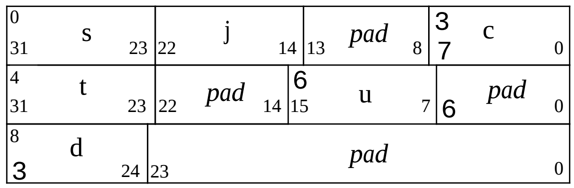


Figure 3-15: Storage Unit Sharing

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

Halfword aligned, sizeof is 2

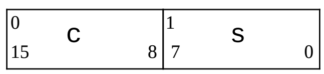


Figure 3-16: `union` Allocation

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

Halfword aligned, sizeof is 2

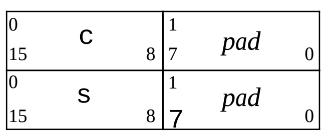
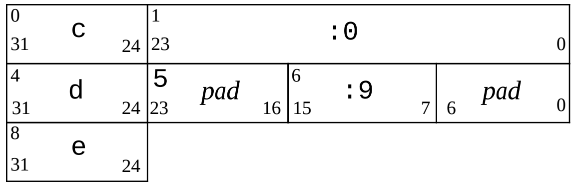


Figure 3-17: Unnamed Bit-Fields

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

Byte aligned, sizeof is 9



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

Function Calling Sequence

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

CPU Registers

The MIPS I ISA specifies 32 general purpose 32-bit registers; two special 32-bit registers that hold the results of multiplication and division instructions; and a 32-bit program counter register. The general registers have the names *$0..$31*. By convention, there is also a set of software names for some of the general registers. Figure 3-18 describes the conventions that constrain register usage. Figure 3-19 describes special CPU registers.

NOTE Not all register usage conventions are described. In particular, register usage conventions in languages other than C are not included, nor are the effects of high optimization levels. These conventions do not affect the interface to the system libraries described in Chapter 6.

Figure 3-18: General CPU Registers

Register Name	Software Name	Use
$0	`zero`	always has the value 0.
$at	`AT`	temporary generally used by assembler.
$2..$3	`v0–v1`	used for expression evaluations and to hold the integer and pointer type function return values.
$4..$7	`a0–a3`	used for passing arguments to functions; values are not preserved across function calls. Additional arguments are passed on the stack, as described below.
$8-$15	`t0–t7`	temporary registers used for expression evaluation; values are not preserved across function calls.
$16-$23	`s0–s7`	saved registers; values are preserved across function calls.
$24..$25	`t8–t9`	temporary registers used for expression evaluations; values are not preserved across function calls. When calling position independent functions *$25* must contain the address of the called function.
$26-$27	`kt0–kt1`	used only by the operating system.
$28 or $gp	`gp`	global pointer and context pointer.
$29 or $sp	`sp`	stack pointer.
$30	`s8`	saved register (like `s0-s7`).
$31	`ra`	return address. The return address is the location to which a function should return control.

Figure 3-19: Special CPU Registers

Register Name	Use
pc	program counter
hi	multiply / divide special register. Holds the most significant 32 bits of multiply or the remainder of a divide
lo	multiply / divide special register. Holds the least significant 32 bits of multiply or the quotient of a divide

NOTE Only registers *$16..$23* and registers *$28.$30* are preserved across a function call. Register *$28* is not preserved, however, when calling position independent code.

Floating–Point Registers

The MIPS ISA provides instruction encodings to move, load, and store values for up to four co-processors. Only co-processor 1 is specified in a *MIPS ABI* compliant system; the effect of moves, loads and stores to the other co-processors (0, 2, and 3) is unspecified.

Co-processor 1 adds 32 32-bit floating-point general registers and a 32-bit control / status register. Each even / odd pair of the 32 floating-point general registers can be used as either a 32-bit single-precision floating-point register or as a 64-bit double-precision floating-point register. For single-precision values, the even-numbered floating-point register holds the value. For double-precision values, the even-numbered floating-point register holds the least significant 32 bits of the value and the odd-numbered floating-point register holds the most significant 32 bits of the value. This is always true, regardless of the byte ordering conventions in use (big endian or little endian).

Floating-point data representation is that specified in *IEEE Standard for Binary Floating-Point Arithmetic,* ANSI / IEEE Standard 754-1985.

Figure 3-20 describes the conventions for using the floating-point registers.

Figure 3-20: Floating Point Registers

Register Name	Use
$f0..$f2	used to hold floating-point type function results; single-precision uses *$f0* and double-precision uses the register pair *$f0..$f1*. *$f2..$f3* return values that are not used in any part of this specification.
$f4..$f10	temporary registers.
$f12..$f14	used to pass the first two single- or double-precision actual arguments.
$f16..$f18	temporary registers.
$f20..$f30	saved registers; their values are preserved across function calls.
fcr31	control / status register. Contains control and status data for floating-point operations, including arithmetic rounding mode and the enabling of floating-point exceptions; it also indicates floating-point exceptions that occurred in the most recently executed instruction and all floating-point exceptions that have occurred since the register was cleared. This register is read / write and is described more fully in the

NOTE Only registers *$f20.$f30* are preserved across a function call. All other floating-point registers can change across a function call. However, functions that use any of *$f20.$f30* for single-precision operations only must still save and restore the corresponding odd-numbered register since the odd-numbered register contents are left undefined by single-precision operations.

NOTE There are other user visible registers in some implementations of the architecture, but these are explicitly not part of this processor supplement. A program that uses these registers is not *ABI* compliant and its behavior is undefined.

The Stack Frame

Each called function in a program allocates a stack frame on the run-time stack, if necessary. A frame is allocated for each non-leaf function and for each leaf function that requires stack storage. A non-leaf function is one that calls other function(s); a leaf function is one that does not itself make any function calls. Stack frames are allocated on the run-time stack; the stack grows downward from high addresses to low addresses.

Each stack frame has sufficient space allocated for:

- local variables and temporaries.

- saved general registers. Space is allocated only for those registers that need to be saved. For non-leaf function, *$31* must be saved. If any of *$16..$23* or *$29..$31* is changed within the called function, it must be saved in the stack frame before use and restored from the stack frame before return from the function. Registers are saved in numerical order, with higher numbered registers saved in higher memory addresses. The register save area must be doubleword (8 byte) aligned.

- saved floating-point registers. Space is allocated only for those registers that need to be saved. If any of *$f20..$f30* is changed within the called function, it must be saved in the stack frame before use and restored from the stack frame before return from the function. Both even- and odd-numbered registers must be saved and restored, even if only single-precision operations are performed since the single-precision operations leave the odd-numbered register contents undefined. The floating-point register save area must be doubleword (8 byte) aligned.

- function call argument area. In a non-leaf function the maximum number of bytes of arguments used to call other functions from the non-leaf function must be allocated. However, at least four words (16 bytes) must always be reserved, even if the maximum number of arguments to any called function is fewer than four words.

- alignment. Although the architecture requires only word alignment, software convention and the operating system require every stack frame to be doubleword (8 byte) aligned.

A function allocates a stack frame by subtracting the size of the stack frame from *$sp* on entry to the function. This *$sp* adjustment must occur before *$sp* is used within the function and prior to any jump or branch instructions.

Figure 3-21: Stack Frame

Base	Offset	Contents	Frame
		unspecified . . . variable size	*High addresses*
	+16	(if present) incoming arguments passed in stack frame	Previous
old *$sp*	+0	space for incoming arguments 1-4	
		locals and temporaries	
		general register save area	Current
		floating-point register save area	
$sp	+0	argument build area	*Low addresses*

The corresponding restoration of *$sp* at the end of a function must occur after any jump or branch instructions except prior to the jump instruction that returns from the function. It can also occupy the branch delay slot of the jump instruction that returns from the function.

Standard Called Function Rules

By convention, there is a set of rules that must be followed by every function that allocates a stack frame. Following this set of rules ensures that, given an arbitrary program counter, return address register *$31*, and stack pointer, there is a deterministic way of performing stack backtracing. These rules also make possible programs that translate already compiled absolute code into position-independent code. See Coding Examples in this chapter.

Within a function that allocates a stack frame, the following rules must be observed:

- In position-independent code that calculates a new value for the *gp* register, the calculation must occur in the first three instructions of the function. One possible optimization is the total elimination of this calculation; a local function called from within a position-independent module guarantees that the context pointer *gp* already points to the global offset table. The calculation must occur in the first basic block of the function.

- The stack pointer must be adjusted to allocate the stack frame before any other use of the stack pointer register.

- At most, one frame pointer can be used in the function. Use of a frame pointer is identified if the stack pointer value is moved into another register, after the stack pointer has been adjusted to allocate the stack frame. This use of a frame pointer must occur within the first basic block of the function before any branch or jump instructions, or in the delay slot of the first branch or jump instruction in the function.

- There is only one exit from a function that contains a stack adjustment: a jump register instruction that transfers control to the location in the return address register *$31*. This instruction, including the contents of its branch delay slot, mark the end of function.

- The deallocation of the stack frame, which is done by adjusting the stack pointer value, must occur once and in the last basic block of the function. The last basic block of a function includes all of the non control-transfer instructions immediately prior to the function exit, including the branch delay slot.

Argument Passing

Arguments are passed to a function in a combination of integer general registers, floating-point registers, and the stack. The number of arguments, their type, and their relative position in the argument list of the calling function determines the mix of registers and memory used to pass arguments. General registers *$4..$7* and floating-point registers *$f12* and *$f14* pass the first few arguments in registers. Double-precision floating-point arguments are passed in the register pairs *$f12, $f13* and *$f14, $f15*; single-precision floating-point arguments are passed in registers *$f12* and *$f14*.

NOTE These argument passing rules apply only to languages such as C that do not do dynamic stack allocation of structures and arrays. Ada is an example of a language that does dynamic stack allocation of structures and arrays.

In determining which register, if any, an argument goes into, take into account the following considerations:

- All integer-valued arguments are passed as 32-bit words, with signed or unsigned bytes and halfwords expanded (promoted) as necessary.

- If the called function returns a structure or union, the caller passes the address of an area that is large enough to hold the structure to the function in *$4*. The called function copies the returned structure into this area before it returns. This address becomes the first argument to the function for the purposes of argument register allocation and all user arguments are shifted down by one.

- Despite the fact that some or all of the arguments to a function are passed in registers, always allocate space on the stack for all arguments. This stack space should be a structure large enough to contain all the arguments, aligned according to normal structure rules (after promotion and structure return pointer insertion). The locations within the stack frame used for arguments are called the home locations.

- At the call site to a function defined with an ellipsis in its prototype, the normal calling conventions apply up until the first argument corresponding to where the ellipsis occurs in the parameter list. If, in the absence of the prototype, this argument and any following arguments would have been passed in floating-point registers, they are instead passed in integer registers. Arguments passed in integer registers are not affected by the ellipsis.

  This is the case only for calls to functions which have prototypes containing an ellipsis. A function without a prototype or without an ellipsis in a prototype is called using the normal argument passing conventions.

- When the first argument is integral, the remaining arguments are passed in the integer registers.
- Structures are passed as if they were very wide integers with their size rounded up to an integral number of words. The fill bits necessary for rounding up are undefined.
- A structure can be split so a portion is passed in registers and the remainder passed on the stack. In this case, the first words are passed in *$4, $5, $6*, and *$7* as needed, with additional words passed on the stack.
- Unions are considered structures.

The rules that determine which arguments go into registers and which ones must be passed on the stack are most easily explained by considering the list of arguments as a structure, aligned according to normal structure rules. Mapping of this structure into the combination of stack and registers is as follows: up to two leading floating-point arguments can be passed in *$f12* and *$f14*; everything else with a structure offset greater than or equal to 16 is passed on the stack. The remainder of the arguments are passed in *$4..$7* based on their structure offset. Holes left in the structure for alignment are unused, whether in registers or in the stack.

The following examples in Figure 3-22 give a representative sampling of the mix of registers and stack used for passing arguments, where d represents double-precision floating-point values, s represents single-precision floating-point values, and n represents integers or pointers. This list is not exhaustive.

See the section “Variable Argument List” later in this section for more information about variable argument lists.

Figure 3-22: Examples of Argument Passing

Argument List	Register and Stack Assignments
d1, d2	*$f12, $f14*
s1, s2	*$f12, $f14*
s1, d1	*$f12, $f14*
d1, s1	*$f12, $f14*
n1, n2, n3, n4	*$4, $5, $6, $7*
d1, n1, d2	*$f12, $6, stack*
d1, n1, n2	*$f12, $6, $7*
s1, n1, n2	*$f12, $5, $6*
n1, n2, n3, d1	*$4, $5, $6, stack*
n1, n2, n3, s1	*$4, $5, $6, $7*
n1, n2, d1	*$4, $5, ($6, $7)*
n1, d1	*$4, ($6, $7)*
s1, s2, s3, s4	*$f12, $f14, $6, $7*
s1, n1, s2, n2	*$f12, $5, $6, $7*
d1, s1, s2	*$f12, $f14, $6*
s1, s2, d1	*$f12, $f14, ($6, $7)*
n1, s1, n2, s2	*$4, $5, $6, $7*
n1, s1, n2, n3	*$4, $5, $6, $7*
n1, n2, s1, n3	*$4, $5, $6, $7*

In the following examples, an ellipsis appears in the second argument slot.

n1, d1, d2	*$4, ($6, $7), stack*
s1, n1	*$f12, $5*
s1, n1, d1	*$f12, $5, ($6, $7)*
d1, n1	*$f12, f6*
d1, n1, d2	*$f12,$6, stack*

Function Return Values

A function can return no value, an integral or pointer value, a floating-point value (single- or double-precision), or a structure; unions are treated the same as structures.

A function that returns no value (also called procedures or void functions) puts no particular value in any register.

A function that returns an integral or pointer value places its result in register *$2.*

A function that returns a floating-point value places its result in floating-point register *$f0.* Floating-point registers can hold single- or double-precision values.

The caller to a function that returns a structure or a union passes the address of an area large enough to hold the structure in register *$4.* Before the function returns to its caller, it will typically copy the return structure to the area in memory pointed to by *$4*; the function also returns a pointer to the returned structure in register *$2.* Having the caller supply the return object's space allows re-entrancy.

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

Both the calling and the called function 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.

NOTE These rules for function return values apply to languages such as C, but do not necessarily apply to other languages. Ada is one language to which the rules do not apply.

Operating System Interface

Virtual Address Space

Processes execute in a 31-bit virtual address space with addresses from 0 to $2^{31} - 1$. Memory management hardware translates virtual addresses to physical addresses, which hides physical addressing and allows a process to run anywhere in the real memory of the system. 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 smallest units of memory allocation in the system. Page size can vary from one system to another, depending on the processor, memory management unit, and system configuration. Processes can call sysconf(BA_OS) to determine the current page size.

Virtual Address Assignments

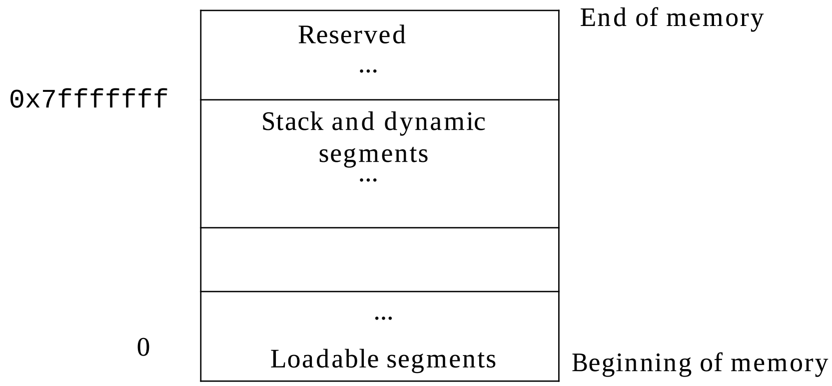
Although processes have the full 31-bit address space available, several factors limit the size of a process.

- The system reserves a configuration-dependent amount of virtual space.
- A tunable configuration parameter limits process size.
- A process that requires more memory than is available in system 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 amount of memory.

Figure 3-23 shows virtual address configuration. The terms used in the figure are:

- The loadable segments of the processes can begin at `0`. The exact addresses depend on the executable file format [see Chapters 4 and 5].
- The stack and dynamic segments reside below the reserved area. Processes can control the amount of virtual memory allotted for stack space, as described below.
- The reserved area resides at the top of virtual space.

Figure 3-23: Virtual Address Configuration



As Figure 3-23 shows, the system reserves the high end of virtual space, with the stack and dynamic segments of a process below that. Although the exact boundary between the reserved area and a process depends on system configuration, the reserved area will not consume more than 4 MBytes from the virtual address space. Thus the user virtual address range has a minimum upper bound of 0x7fbfffff. Individual systems can reserve less space, increasing the processes virtual memory range. More information follows in ‘Managing the Process Stack.’

Although applications can control their memory assignments, the typical arrangement follows the diagram in Figure 3-23. Loadable segments reside at low ad-

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

Process address spaces typically have three segment areas that can change size from one execution to the next: the stack [through setrlimit(BA_OS)], the data segment [through malloc(BA_OS)], and the dynamic segment area [through mmap(KE_OS)]. Changes in one area can 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 uses mmap(KE_OS) to request a mapping at a specific address could work in some environments and fail in others. For this reason, programs that establish a mapping in their address space should use an address provided by the system.

Despite these warnings about requesting specific addresses, the facility can be used properly. For example, a multiprocess application can map several files into the address space of each process and build relative pointers among the data in the files. This is done by having each process specify a certain amount of memory at an address chosen by the system. After each process receives its own address from the system, it can 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 specify addresses, the application cannot build shared data structures, because the *relative* positions for files in each process would be unpredictable.

Exception Interface

In MIPS architecture, there are two execution modes: user and kernel. Processes run in user mode and the operating system kernel runs in kernel mode. The processor changes mode to handle precise or interrupting *exceptions.* Precise exceptions, which result from instruction execution, are explicitly generated by a process. This section, therefore, specifies those exception types with defined behavior.

An exception results in the operating system kernel taking some action. After handling the exception the kernel restarts the user process. It is not possible to determine that an exception took place, except by apparent slower execution. Some exceptions are considered errors, however, and cannot be handled by the operating system kernel. These exceptions cause either process termination or, if signal catching is enabled, send a signal to the user process (see signal(BA_OS)).

Figure 3-24 lists the correspondence between exceptions and the signals specified by signal(BA_OS).

Figure 3-24: Hardware Exceptions and Signals

Exception	Signal
TLB modification	SIGBUS
Read TLB miss	SIGSEGV
Read TLB miss	SIGBUS
Write TLB miss	SIGSEGV
Read Address Error	SIGBUS
Write Address Error	SIGBUS
Instruction Bus Error	SIGBUS
Data Bus Error	SIGBUS
Syscall	SIGSYS
Breakpoint	SIGTRAP
Reserved Instruction	SIGILL
Coprocessor Unusable	SIGILL
Arithmetic Overflow	SIGFPE

NOTE A Read TLB miss generates a SIGSEGV signal when unmapped memory is accessed. A Read TLB miss generates a SIGBUS signal when mapped, but otherwise inaccessible memory is accessed. In other words, a SIGBUS is generated on a protection fault while a SIGSEGV is generated on a segmentation fault.

Floating-point instructions exist in the architecture, and can be implemented either in hardware or software. If the Coprocessor Unusable exception occurs because of a coprocessor 1 instruction, the process receives no signal. Instead, the system intercepts the exception, emulates the instruction, and returns control to the process. A process receives SIGILL for the Coprocessor Unusable exception only when the accessed coprocessor is not present and when it is not coprocessor 1.

System calls, or requests for operating system services, use the Syscall exception for low level implementation. Normally, system calls do not generate a signal, but SIGSYS can occur in some error conditions.

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

Stack Backtracing

There are standard called function rules for functions that allocate a stack frame and because the operating system kernel initializes the return address register *$31* to zero when starting a user program it is possible to trace back through any arbitrarily nested function calls. The following algorithm, which takes the set of general registers plus the program counter as input, produces the values the registers had at the most recent function call. Of course, only the saved registers plus *gp, sp, ra,* and *pc* can be reconstructed.

- Scan each instruction starting at the current program counter, going backwards. The compiler and linker must guarantee that a jump register to return address instruction will always precede each text section.

  - If the instruction is of the form "move *$r, sp*" or "addu *$r, $sp, $0,* then the register *$r* may be a frame pointer. The algorithm remembers the current instruction so it can continue its backward scan.

    Then, it scans forward until it sees the "jr *ra*" instruction that marks the end of the current function.

    Next, it scans backwards searching for an instruction of the form "move *sp, $r*" or "addu *$sp, $r, $0*". This scan terminates when such an instruction is found or the branch or jump instruction that marks the beginning of the last basic block.

    If a move or addu instruction of the kind described above was found, remember the register number of *$r* as the frame pointer. Otherwise, *$r* is not the frame pointer.

    The algorithm should return to its original backwards scan starting with the instruction preceding the one remembered above.

  - If the instruction is a stack pointer decrement, exit the scan.

  - If the instruction is a jump register to return address, exit the scan.

- If the last examined instruction is a jump register to the return address, it is the end of the previous function and no stack frame has yet been allocated for the current function. The address from which the current function was called is in the return address register minus eight. The other save registers had their current values when this function was called, so just return their current values.

- The stack decrement instruction must occur in the first basic block of the function. The amount of stack decrement is the size of the stack frame.

- Examine each instruction at increasing program addresses. If any instruction is a store of save registers *$16-$23, $28, $30,* or *$31* through the frame pointer (or stack pointer if no frame pointer was used), then record its value by reading from the stack frame.

- Stop after examining the instruction in the first branch delay slot encountered. This marks the end of the first basic block.

- The frame pointer is the stack pointer value at the time the current function was called (or the stack pointer if no frame pointer was used) plus the size of the stack frame.

- The address from which the function is called is either the return address register value minus eight or, if the return address was saved on the stack, the saved value minus eight.

Process Initialization

This section describes the machine state that exec(BA_OS) creates for “infant” processes, including argument passing, register usage, stack frame layout, etc. Programming language systems use this initial program state to establish a standard environment for their application programs. For example, a C program begins execution at a function named main, conventionally declared as follows:

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

where 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 does provide the information necessary to implement a call to main or to the entry point for a program in any other language.

Special Registers

As the architecture defines, two registers control and monitor the processor: the status register (SR) and the floating-point control and status register (csr). Applications cannot access the SR directly; they run in *user mode*. Instructions to read and write the SR are privileged. No fields in the SR affect user program behavior, except that the program can assume that coprocessor 1 instructions work as documented and that the user program executes in user mode with the possibility that interrupts are enabled. Nothing more should be inferred about the contents of the SR.

Figure 3-25 lists the initial values of the floating-point control and status register provided in the architecture

Figure 3-25: Floating–Point Control and Status Register Fields

Field	Value	Note
C	0	Condition
Bit Exceptions	0	No current exceptions
Trap Enables	0	Floating-point traps not enabled
Sticky Bits	0	No accrued exceptions
RM	0	Round to nearest

The *ABI* specifies that coprocessor 1 always exists and that coprocessor 1 instructions (floating-point instructions) work as documented. Programs that directly execute coprocessor 0, 2, or 3 instructions do not conform to the *ABI*. Individual system implementations may use one of these coprocessors under control of the *system* software, not the application.

Process Stack

When a process receives control, its stack holds the arguments and environment from exec(BA_OS).Figure 3-26 shows the initial process stack.

Figure 3-26: Initial Process Stack

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

Argument strings, environment strings, and auxiliary information do not appear in a specific order with the information block. The system may leave an unspecified amount of memory between a null auxiliary vector entry and the beginning of an information block.

Except as shown below, general integer and floating-point register values are unspecified at process entry. Consequently, a program that requires specific register values must set them explicitly during process initialization. It should *not* rely on the operating system to set all registers to 0.

The registers listed below have the specified contents at process entry:

$2 A non-zero value specifies a function pointer the application should register with `atexit`(BA_OS). If *$2* contains zero, no action is required.

$sp The stack pointer holds the address of the bottom of the stack, which must be doubleword (8 byte) aligned.

$31 The return address register is set to zero so that programs that search backward through stack frames (stack backtracing) recognize the last stack frame, that is, a stack frame with a zero in the saved *$31* slot.

Every process has a stack, but the system does not define a 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 structures shown in Figure 3-27, interpreted according to the a_type member.

Figure 3-27: Auxillary Vector

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

Figure 3-28: Auxillary Vector Types, `a_type`

Name	Value	a_un
AT_NULL	0	ignored
AT_IGNORE	1	ignored
AT_EXECFD	2	a_val
AT_PHDR	3	a_ptr
AT_PHENT	4	a_val
AT_PHNUM	5	a_val
AT_PAGESZ	6	a_val
AT_BASE	7	a_ptr
AT_FLAGS	8	a_val
AT_ENTRY	9	a_ptr
AT_NOTELF	10	a_val
AT_UID	11	a_val
AT_EUID	12	a_val
AT_GID	13	a_val
AT_EGID	14	a_val

The auxiliary vector types (a_type) shown in Figure 3-28 are explained in the paragraphs below:

`AT_NULL` The auxiliary vector has no fixed length; instead the `a_type` member of the last entry 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) can pass control to an interpreter program. When this happens, the system places either an entry of type `AT_EXECFD` or 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 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 are also 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, the a_val member of this entry 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.

AT_NOTELF — The a_val member of this entry is zero if the executable is in ELF format as described in Chapter 4. It is non-zero if the executable is in MIPS XCOFF format.

AT_UID — If present, the a_val member of this entry holds the actual user id of the current user.

AT_EUID — If present, the a_val member of this entry holds the effective user id of the current user.

AT_GID — If present, the a_val member of this entry holds the actual group id of the current user.

AT_EGID — If present, the a_val member of this entry holds the effective group id of the current user.

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 0x7fc00000, growing toward lower addresses. The process receives three arguments:

- cp
- src
- dst

It also inherits two environment strings. (The example does not 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 (8 byte) alignment.

Figure 3-29: Example Process Stack

Address					
	n	:	\0	*pad*	*High addresses*
	r	/	b	i	
	:	/	u	s	
0x7fbffff0	/	b	i	n	
	/	d	i	r	
	h	o	m	e	
	T	H	=	/	
0x7fbfffe0	r	\0	P	A	
	e	/	d	i	
	/	h	o	m	
	O	M	E	=	
0x7fbfffd0	s	t	\0	H	
	r	c	\0	d	
	c	p	\0	s	
	0				
0x7fbfffc0	0				
	13				
	2				Auxiliary vector
	0				
0x7fbfffb0	0x7fbfffe2				
	0x7fbfffd3				Environment vector
	0				
	0x7fbfffcf				
0x7fbfffa0	0x7fbfffcb				
	0x7fbfffc8				Argument vector
$sp+0 0x7fbfff90	3				Argument count
					Low addresses

Coding Examples

This section discusses example code sequences for basic 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 uses the machine or theoperating system, and specify what a program can or cannot assume about the execution environment. Unlike the previous material, the information here illustrates how operations *can* 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 absolute addresses coincide with the process 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.

The following sections describe the differences between absolute code and position-independent code. Code sequences for the models (when different) appear together, allowing easier comparison

NOTE The examples below show code fragments with various simplifications. They are intended to explain addressing modes, not to show optimal code sequences or to reproduce compiler output or actual assembler syntax.

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

Code Model Overview

When the system creates a process image, the executable file portion of the process has fixed addresses, and the system chooses shared object library virtual addresses to avoid conflicts with other segments in the process. To maximize text sharing, shared objects conventionally use position-independent code, in which instructions contain no absolute addresses. Shared object text segments can be loaded at various virtual addresses without changing 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. If the target location exceeds the allowable offset for PC-relative addressing, the program requires an absolute address.

- When the program requires an absolute address, it computes the desired value. Instead of embedding absolute addresses in the 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 easily satisfy the first condition.

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 assigned for an individual process. Because data segments are private for each process, the table entries can change - whereas text segments do not change because multiple processes share them.

Due to the 16-bit *offset* field of load and store instructions, the global offset table is limited to 16,384 entries (65,536 bytes).

The 16-bit offset fields of instructions require two instructions to load a 32-bit absolute value into a register. In the following code fragments wherever a 32-bit abso

lute value is loaded with a combination of lui and addiu instructions, the proper correction was made to the high 16 bits before setting the most significant (sign) bit of the low order 16 bits of the value.

Position–Independent Function Prologue

This section describes the function prologue for position-independent code. A function prologue first calculates the address of the global offset table, leaving the value in register *$28*, hereafter referred to by its software name gp. This address is also known as the *context pointer*. This calculation is a constant offset between the text and data segments, known at the time the program is linked.

The offset between the start of a function and the global offset table (known because the global offset table is kept in the data segment) is added to the virtual address of the function to derive the virtual address of the global offset table. This value is maintained in the *gp* register throughout the function.

The virtual address of a called function is passed to the function in general register *$25*, hereafter referred to by its software name t9. All callers of position independent functions must place the address of the called function in t9.

NOTE Although this section contains examples, an *ABI* compliant program must use register `t9` for the context register. The interface to the system library routines described in Chapter 6 of the *System V ABI* relies on the address of the called procedure being passed in `t9`.

After calculating the *gp*, a function allocates the local stack space and saves the gp on the stack, so it can be restored after subsequent function calls. In other words, the gp is a *caller saved* register.

The code in the following figure illustrates a position-independent function prologue. _gp_disp represents the offset between the beginning of the function and the global offset table.

```
name:
   la     gp, _gp_disp
   addu   gp, gp, t9
   addiu  sp, sp, –64
   sw     gp, 32(sp)
```

Various optimizations are possible in this code example and the others that follow. For example, the calculation of *gp* need not be done for a position-independent function that is strictly local to an object module. However, the simplest, most general examples are used to keep the complexity to a minimum.

Data Objects

This section describes data objects with static storage duration. The discussion excludes stack-resident objects, because programs always compute their virtual addresses relative to the stack pointer.

In the MIPS architecture, only load and store instructions access memory. Because instructions cannot directly hold 32-bit addresses, a program normally computes an address into a register, using one instruction to load the high 16 bits of the address and another instruction to add the low 16 bits of the address.

NOTE In actual practice, most data references are performed by a single machine instruction using a gp relative address into the *global data area* (the global offset table and the global data area are both addressed by gp in position–independent code). However, those references are already position–independent and this section illustrates the differences between absolute addressing and position independent addressing.

Figure 3-30: Absolute Load and Store

C	Assembly

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

`ptr = &dst;`	`lui     t6, dst >> 16`
	`addiu   t6, t6, dst & 0xffff`
	`lui     t7, ptr >> 16`
	`sw      t6, ptr & 0xffff(t7)`

`*ptr = src;`	`lui     t6, src >> 16`
	`lw      t6, src & 0xffff(t6)`
	`lui     t7, ptr >> 16`
	`lw      t7, ptr & 0xffff(t7)`
	`sw      t6, 0(t7)`

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 *gp* gives the absolute address of the table entry holding the desired address .

NOTE The offset of data item *name* is represented as *name_got_off* in the global offset table. This is only a convention and there is no actual assembler support for these constructs.

Position-Independent Load and Store

C	Assembly

extern int src;
extern int dst;
extern int *ptr;
ptr = &dst;
 	
*ptr = src;
	globl src, dst, ptr
	
	lw t7, dst_got_off(gp)
	lw t6, ptr_got_off(gp)
	nop
	sw t7, 0(t6)
	lw t7, src_got_off(gp)
	nop
	lw t7, 0(t7)
	lw t6, ptr_got_off(gp)
	nop
	lw t6, 0(t6)
	nop
	sw t7, 0(t6)

Function Calls

Programs use the jump and link instruction, jal, to make direct function calls. Since the jal instruction provides 28 bits of address and the program counter contributes the four most significant bits, direct function calls are limited to the current 256 MByte chunk of the address space as defined by the four most significant bits of pc.

Figure 3-31: Absolute Direct Function Call

C	Assembly
extern void function(); function();	jal function nop

```
C:
extern void function();
function();

Assembly:
jal         function
nop
```

Calls to functions outside the 256 MByte range and other indirect function calls are done by computing the address of the called function into a register and using the jump and link register, jalr , instruction.

Figure 3-32: Absolute Indirect Function Call

```
C                              Assembly

extern void (*ptr)();
extern void name();
ptr = name;                    lui     t6, name >> 16
                               addiu   t6, t6, name & 0xffff
                               lui     t7, ptr >> 16
                               sw      t6, ptr & 0xffff(t7)
(*ptr)();                      lui     t6, ptr >> 16
                               addiu   t6, t6, ptr & 0xffff
                               jalr    ra, t6
                               nop
```

NOTE Normally, the data area for the variable `ptr` is kept in the global data area and is accessed relative to register `gp`. However, this example illustrates the difference between absolute data references and position–independent data references.

Calling position independent code functions is always done with the jalr instruction. The global offset table holds the absolute addresses of all position independent functions.

Figure 3-33: Position-Independent Function Calls

C	Assembly

`extern void (*ptr)();`
`extern void name();`
`name();`
	`.global ptr, name`
	
	`lw t9, name_got_off(gp)`
	`nop`
	`jalr t9`
	`nop`
	`lw gp, 24(sp)`
	`nop`

`ptr = name;`	`lw t7, name_got_off(gp)`
	`lw t6, ptr_got_off(gp)`
	`nop`
	`sw t7,0(t6)`

`(*ptr)();`	`lw t7, ptr_got_off(gp)`
	`nop`
	`lw t9, 0(t7)`
	`nop`
	`jalr t9`
	`nop`
	`lw gp, 24(sp)`
	`nop`

NOTE gp must be restored on return because called position independent functions can change it. gp is saved in the stack frame in the prologue of position–independent code functions.

Branching

Programs use branch instructions to control execution flow. As defined by the architecture, branch instructions hold a PC-relative value with a 256 KByte range, allowing a jump to locations up to 128 KBytes away in either direction.

Figure 3-34: Branch Instruction, All Models

C	Assembly

`label:`
`    .  .  .`
`    goto label;`
	`$32:`
	`.  .  .`
	`b     $32`
	`nop`

C switch statements provide multiway selection. When case labels of a switch statement satisfy grouping constraints, the compiler implements the selection with an address table. The address table is placed in a .rdata section; this so the linker can properly relocate the entries in the address table. Figures 3-36 and 3-37 use the following conventions to hide irrelevant details:

- The selection expression resides in register `t7;`
- `case` label constants begin at zero;
- `case` labels, `default`, and the address table use assembly names `.Lcase`*i*, `.Ldef`, and `.Ltab,` respectively.

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

Figure 3-35: Absolute switch Code

C	Assembly

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

```
               sltiu     at, t7, 4
               beq       at, zero, .Ldef
               sll       t7, t7, 2
               lui       t6, .Ltab >> 16
               addiu     t6, .Ltab & 0xffff
               addu      t6, t6, t7
               lw        t7, 0(t6)
               nop
               jr        t7
               nop
.Ltab:         .word     .Lcase0
               .word     .Ldef
               .word     .Lcase2
               .word     .Lcase3
```

Figure 3-36: Position-independent switch Code

C

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

Assembly

```
            sltiu    at, t7, 4
            beq      at, zero, .Ldef
            sll      t7, t7, 2
            lw       at, .Ltab_got_off(gp)
            nop
            addu     at, at, t7
            lw       t6, 0(at)
            nop
            addu     t6, t6, gp
            jr       t6
            nop
            .rdata
.Ltab:      .word    .Lcase0_gp_off
            .word    .Ldef_gp_off
            .word    .Lcase2_gp_off
            .word    .Lcase3_gp_off
```

C Stack Frame

Figure 3-37 shows the C stack frame organization.It conforms to the standard stack frame with designated roles for unspecified areas in standard frame.

Figure 3-37: C Stack Frame

Base	Offset	Contents	Frame
		local space: automatic variables . . . compiler scratch space: temporaries register save area	*High addresses* Current
$sp	16	outgoing arguments 5	
		outgoing argument 4 . . .	
$sp	0	outgoing argument 1	*Low addresses*

A C stack frame does not 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 positive offsets from *sp*. Its incoming arguments reside in the previous frame, referenced as positive offsets from *sp* plus the size of the stack frame.

Variable Argument List

Previous sections describe the rules for passing arguments. Unfortunately, some otherwise portable C programs depend on other argument passing schemes, 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 *no*t work on MIPS based systems because some arguments can 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 MIPS and other machines as well). A program implicitly uses <stdarg.h> when it specifies a prototype declaration with an ellipsis ("...") in the argument list. No prototype or a prototype with no ellipsis causes <varargs.h> to be used.

When a function uses <stdarg.h>, the compiler modifies the argument passing rules described above. In the calling function, the compiler passes the first 4 32-bit words of arguments in registers *$4, $5, $*6, and *$7,* regardless of data type. In particular, this means that floats and doubles are passed in the integer register. In the called function, the compiler arranges that the argument registers are saved on the stack in the locations reserved for incoming arguments. This allows the called function to reference all incoming arguments from consecutive locations on the stack.

When a function uses <varargs.h>, the situation is somewhat different. The calling function uses the argument passing rules exactly as described in the the section on argument passing rules. However, the called function allocates 32 bytes immediately adjacent to the space for incoming arguments in which to save incoming floating-point argument values.

If va_list appears as the first argument, it spills the $*f12/$f13,* and *$f14/$f15* register pairs at -24 and -32 bytes respectively, relative to the increasing argument area. If `va_alist` appears as the second argument, it spills the $f14/$f15 register pair at -24 bytes relative to the incoming argument area.

Figure 3-38: Called Function Stack Frame

Base	Offset	Contents	Frame
		unspecified . . . variable size	*High addresses*
	+16	(if present) incoming arguments passed in stack frame	Previous
old *$sp*	+0	space for incoming arguments 1-4	
		16 bytes reserved 8 bytes to spill *$f12/$f13* 8 bytes to spill *$f14/$f15*	
		locals and temporaries	
		general register save area	Current
		floating-point register save area	
$sp	+0	argument build area	*Low addresses*

The 30 most-significant bits of the va_list type locate the next address in the incoming arguments to process with the va_arg macro. This address is calculated by the rules given below. The two least significant bits encode whether the va_arg macro will read floating-point values from the incoming argument area or from the floating-point save area described in the previous paragraph.

The va_start() macro in <varargs.h> encodes the following states in the two least significant bits of the va_list type:

- If the `va_list` pointer points to the first argument, `va_start` subtracts 1 from the `va_list` pointer, leaving it completely misaligned.
- If the `va_list` pointer points to the second argument, and the first argument was type double, `va_start` subtracts 2 from the `va_list` pointer, leaving it 2-byte aligned.
- For all other cases, `va_start` leaves the low-order bits of the `va_list` pointer set to zero (leaving it 4-byte aligned).

The va_start() macro in <varargs.h> requires built-in compiler support to determine which position in the argument list the va_alist parameter appears.

The va_start()macro in <stdarg.h> always sets the two least significant bits of the va_list type to zero.

If the second argument of the va_arg() macro is not the type double or the va_list pointer is 4-byte aligned, it zeroes the two least significant bits of the va_list pointer in calculating the next argument to return. It advances the value of the va_list pointer by the size of the type passed to va_arg. This leaves the va_list pointer 4-byte aligned.

If the second argument to va_arg() is type double and the va_list pointer's least significant bit is 1, it returns the value of the *$f12/$f13* register pair saved 32 bytes below the incoming argument. The address of the save area must be calculated by subtracting 31 from the value of the va_list pointer. The va_arg macro advances va_list pointer by 7 leaving it 2-byte aligned.

If the second argument to va_arg() is type double and the va_list pointer's value is 2-byte aligned, it returns the value of the *$f14/$f15* register pair saved 16 bytes below the incoming argument area. The address of the save area must be calculated by subtracting -30 from the value of the va_list pointer. The va_arg macro advances va_list pointer by 10 leaving it 4-byte aligned.

Dynamic Allocation of Stack Space

The C language does not require dynamic stack allocation *within* a stack frame. Frames are allocated dynamically on the program stack, depending on program execution. The architecture, standard calling sequence, and stack frame support dynamic allocation for programming languages that require it. Thus languages that need dynamic stack frame sizes can call C functions and vice versa.

When a function requires dynamically allocated stack space it manifests a *frame pointer* on entry to the function. The frame pointer is kept in a callee-saved register so that it is not changed across subsequent function calls. Dynamic stack allocation requires the following steps.

1. On function entry, the function adjusts the stack pointer by the size of the static stack frame. The frame pointer is then set to this initial *sp* value and is used for referencing the static elements within the stack frame, performing the normal function of the stack pointer.

2. Stack frames are doubleword (8 byte) aligned; dynamic allocation preserves this property. Thus, the program rounds (up) the desired byte count to a multiple of 8.

3. To allocate dynamic stack space, the program decreases the stack pointer by the rounded byte count, increasing its frame size. At this point, the new space resides between the register save area and the argument build area and the argument build area effectively moves down.

NOTE Standard calling sequence rules require that any frame pointer manifest within a function be initialized within the first basic block of the function. In other words, it must be set before any branches or calls.

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 resumes the dynamic allocation from the point of interruption.

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

- The process can terminate.

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

Existing stack objects reside at fixed offsets from the frame pointer; stack heap allocation does not move them. No special code is needed to free dynamically allocated stack memory. The function epilogue 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[], MIPS requires the values listed in Figure 4-1.

Figure 4–1: MIPS Identification, e_ident[]

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

Processor identification resides in the ELF header e_machine member and must have the value 8, defined as the name EM_MIPS.

The ELF header e_flags member holds bit flags associated with the file, as listed in Figure 4-2.

Figure 4–2: Processor–Specific Flags, e_flags

Name	Value
EF_MIPS_NOREORDER	0x00000001
EF_MIPS_PIC	0x00000002
EF_MIPS_CPIC	0x00000004
EF_MIPS_ARCH	0xf0000000

EF_MIPS_NOREORDER
: This bit is asserted when at least one .noreorder directive in an assembly language source contributes to the object module.

EF_MIPS_PIC
: This bit is asserted when the file contains position-independent code that can be relocated in memory.

EF_MIPS_CPIC
: This bit is asserted when the file contains code that follows standard calling sequence rules for calling position-independent code. The code in this file is not necessarily position independent. The EF_MIPS_PIC and EF_MIPS_CPIC flags

must be mutually exclusive.

EF_MIPS_ARCH — The integer value formed by these four bits identify extensions to the basic MIPS I architecture. An *ABI* compliant file must have the value zero in these four bits. Non-zero values indicate the object file or executable contains program text that uses architectural extensions to the MIPS I architecture.

Sections

Figure 4-3 lists the MIPS-defined special section index which is provided in addition to the standard special section indexes.

Figure 4–3: Special Section Indexes

Name	Value
SHN_MIPS_ACOMMON	0xff00 or (SHN_LOPROC + 0)
SHN_MIPS_TEXT	0xff01 or (SHN_LOPROC + 1)
SHN_MIPS_DATA	0xff02 or (SHN_LOPROC + 2)
SHN_MIPS_ SCOMMON	0xff03 or (SHN_LOPROC + 3)
SHN_MIPS_SUNDEFINED	0xff04 or (SHN_LOPROC + 4)

SHN_MIPS_ACOMMON
: Symbols defined relative to this section are common symbols which are defined and allocated. The st_value member of such a symbol contains the virtual address for that symbol. If the section must be relocated, the alignment indicated by the virtual address is preserved, up to modulo 65,536. Symbols found in shared objects with section index SHN_COMMON are not allocated in the shared object. The dynamic linker must allocate space for SHN_COMMON symbols that do not resolve to a defined symbol.

SHN_MIPS_TEXT

SHN_MIPS_DATA
: Symbols defined relative to these two sections are only present after a program has been rewritten by the pixie code profiling program. Such rewritten programs are not ABI-compliant. Symbols defined relative to these two sections will never occur in an ABI-compliant program.

SHN_MIPS_SCOMMON
: Symbols defined relative to this section are common symbols which can be placed in the global data area (are *gp*-addressable). See "Global Data Area" in this chapter. This section only occurs in relocatable object files.

SHN_MIPS_SUNDEFINED — Undefined symbols with this special section index in the *st_shndx* field can be placed in the global data area (*gp*-addressable). See "Global Data Area" in this chapter. This section only occurs in relocatable object files.

Figure 4-4 lists the MIPS-defined section types in addition to the standard section types.

Figure 4–4: Section Types, `sh_type`

Name	Value
`SHT_MIPS_LIBLIST`	`0x70000000 or (SHT_LOPROC + 0)`
`SHT_MIPS_CONFLICT`	`0x70000002 or (SHT_LOPROC + 2)`
`SHT_MIPS_GPTAB`	`0x70000003 or (SHT_LOPROC + 3)`
`SHT_MIPS_UCODE`	`0x70000004 or (SHT_LOPROC + 4)`
`SHT_MIPS_DEBUG`	`0x70000005 or (SHT_LOPROC + 5)`
`SHT_MIPS_REGINFO`	`0x70000006 or (SHT_LOPROC + 6)`

SHT_MIPS_LIBLIST — The section contains information about the set of dynamic shared object libraries used when statically linking a program. Each entry contains information such as the library name, timestamp, and version. See "Quickstart" in Chapter 5 for details.

SHT_MIPS_CONFLICT — The section contains a list of symbols in an executable whose definitions conflict with shared-object defined symbols. See "Quickstart" in Chapter 5 for details.

SHT_MIPS_GPTAB — The section contains the *global pointer table*. The global pointer table includes a list of possible global data area sizes. The list allows the linker to provide the user with information on the optimal size criteria to use for gp register relative addressing. See "Global Data Area" below for details.

SHT_MIPS_UCODE — This section type is reserved and the contents are unspecified. The section contents can be ignored.

SHT_MIPS_DEBUG — The section contains debug information specific to MIPS. An ABI-compliant application does not need to have a section of this type.

SHT_MIPS_REGINFO — The section contains information regarding register usage information for the object file. See Register Information for details.

A section header sh_flags member holds 1-bit flags that describe the attributes of the section. In addition to the values defined in the *System V ABI,* Figure 4-5 lists the MIPS-defined flag.

Figure 4–5: Section Attribute Flags, sh_flags

Name	Value
SHF_MIPS_GPREL	0x10000000

SHF_MIPS_GPREL — The section contains data that must be part of the global data area during program execution. Data in this area is addressable with a gp relative address. Any section with the SHF_MIPS_GPREL attribute must have a section header index of one of the .gptab special sections in the sh_link member of its section header table entry. See "Global Data Area" below for details.

The static linker does not guarantee that a section with the SHF_MIPS_GPREL attribute will remain in the global data area after static linking.

Figure 4-6 lists the MIPS-defined section header sh_link and sh_info members interpretation for the MIPS-specific section types.

Figure 4–6: `sh_link` and `sh_info` interpretation

sh_type	sh_link	sh_info
SHT_MIPS_LIBLIST	The section header index of the string table used by entries in this section.	The number of entries in this section.
SHT_MIPS_GPTAB	not used	The section header index of the SHF_ALLOC + SHF_WRITE section. See " Global Data Area" in this chapter.

Special Sections

MIPS defines several additional special sections. Figure 4-7 lists their types and corresponding attributes.

Figure 4–7: Special Sections

Name	Type	Attributes
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.sdata	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE + \ SHF_MIPS_GPREL
.sbss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE + \ SHF_MIPS_GPREL
.lit4	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE + \ SHF_MIPS_GPREL
.lit8	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE + \ SHF_MIPS_GPREL
.reginfo	SHT_MIPS_REGINFO	SHF_ALLOC
.liblist	SHT_MIPS_LIBLIST	SHF_ALLOC
.conflict	SHT_CONFLICT	SHF_ALLOC
.gptab	SHT_MIPS_GPTAB	none
.got	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE + \ SHF_MIPS_GPREL
.ucode	SHT_MIPS_UCODE	none
.mdebug	SHT_MIPS_DEBUG	none
.dynamic	SHT_DYNAMIC	SHF_ALLOC
.rel.dyn	SHT_REL	SHF_ALLOC

NOTE A *MIPS ABI* compliant system must support the `.sdata`, `.sbss`, `.lit4`, `.lit8`, `.reginfo`, and `.gptab` sections. A *MIPS ABI* compliant system must recognize, but may choose to ignore the contents of the `.liblist or .conflict` sections. However, if either of these optional sections is supported, both must be supported.

`.text` This section contains only executable instructions. The first two instructions immediately preceding the first function in the section must be a jump to return address instruction followed by a nop. The stack traceback algorithm, described in Chapter 3, depends on this.

`.sdata` This section holds initialized short data that contribute to the program memory image. See "Global Data Area" below for details.

.sbss	This section holds uninitialized short data that contribute to the program memory image. By definition, the system initializes the data with zeros when the program begins to run. See "Global Data Area" below for details.
.lit4	This section holds 4 byte read-only literals that contribute to the program memory image. Its purpose is to provide a list of unique 4-byte literals used by a program. See "Global Data Area" below for details. Although this section has the SHF_WRITE attribute, it is not expected to be written. Placing this section in the data segment mandates the SHF_WRITE attribute.
.lit8	This section holds 8 byte read-only literals that contribute to the program memory image. Its purpose is to provide a list of unique 8-byte literals used by a program. See "Global Data Area" below for details. Although this section has the SHF_WRITE attribute, it is not expected to be written. Placing this section in the data segment mandates the SHF_WRITE attribute.
.reginfo	This section provides information on the program register usage to the system. See "Register Information" below for details.
.liblist	This section contains information on each of the libraries used at static link time as described in "Quickstart" in Chapter 5.
.conflict	This section provides additional dynamic linking information about symbols in an executable file that conflict with symbols defined in the dynamic shared libraries with which the file is linked. See "Quickstart" in Chapter 5 for details.
.gptab	This section contains a global pointer table. The global pointer table is described in "Global Data Area" in this chapter. The section is named .gptab.sbss,.gptab.sdata, gptab.bss, or .gptab.data depending on which data section the particular .gptab refers.
.ucode	This section name is reserved and the contents of this type of section are unspecified. The section contents can be ignored

`.mdebug` This section contains symbol table information as emitted by the MIPS compilers. Its content is described in Chapter 10 of the *MIPS Assembly Language Programmer's Guide*, order number ASM-01-DOC, (Copyright © 1989, MIPS Computer Systems, Inc.). The information in this section is dependent on the location of other sections in the file; if an object is relocated, the section must be updated. Discard this section if an object file is relocated and the *ABI* compliant system does not update the section.

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

`.dynamic` This is the same as the generic ABI section of the same type, but the MIPS-specific version does not include the `SHF_WRITE` attribute.

`.rel.dyn` This relocation section contains run-time entries for the `.data` and `.sdata` sections. See "Relocations" in Chapter 5 for more information.

NOTE Sections that contribute to a loadable program segment must not contain overlapping virtual addresses.

Symbol Table

Symbol Values

If an executable or shared object 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 signals to the dynamic linker that the symbol definition for that function is not contained in the executable file. If there is a stub for that symbol in the executable file and the st_value member for the symbol table entry is non-zero, the value will contain the virtual address of the first instruction of that procedure's stub. Otherwise, the st_value member contains zero. This stub calls the dynamic linker at runtime for lazy text evaluation. See "Function Addresses" in Chapter 5 for details.

Global Data Area

The global data area is part of the data segment of an executable program. It con-

Figure 4–8: Global Pointer Table

```
typedef union {
      struct {
            Elf32_Word     gt_current_g_value;
            Elf32_Word     gt_unused;
      } gt_header;
      struct {
            Elf32_Word     gt_g_value;
            Elf32_Word     gt_bytes;
      } gt_entry;
 } Elf32_gptab;
```

`gt_header.gt_current_g_value`
: This member is the size criterion actually used for this object file. Data items of this size or smaller are referenced with gp relative addressing and reside in a `SHF_MIPS_GPREL` section.

`gt_header.gt_unused`
: This member is not used in the first entry of the `Elf32_gptab` array.

`gt_entry.gt_g_value`
: This member is a hypothetical size criterion value.

`gt_entry.gt_bytes`
: This member indicates the length of the global data area if the corresponding `gt_entry.gt_g_value` were used.

The first element of the `ELF_32_gptab` array is alway of type `gt_header`; this entry must always exist. Additional elements of the array are of type `gt_entry`.

Each of the `gt_entry.gt_g_value` fields is the size of an actual data item encountered during compilation or assembly, including zero. Each separate size criteria results in a overall size for the global data area. The various entries are sorted and duplicates are removed. The resulting set of entries, including the actual size criterion used, yields the `.gptab` section.

There are always at least two `.gptab`

Register Information

The compilers and assembler collect information on the registers used by the code in the object file. This information is communicated to the operating system kernel using a `.reginfo` section. The operating system kernel can use this information to decide what registers it does not need to save or which coprocessors the program uses. The section also contains a field which specifies the initial value for the `gp` register, based on the final location of the global data area in memory.

Figure 4-9: Register Information Structure

```
typedef struct {
   Elf32_Word     ri_gprmask;;
   Elf32_Word     ri_cprmask[4];
   Elf32_SWord    ri_gp_value;
 } ELF_RegInfo;
```

`ri_gprmask` This member contains a bit-mask of general registers used by the program. Each set bit indicates a general integer register used by the program. Each clear bit indicates a general integer register not used by the program. For instance, bit 31 set indicates register *$31* is used by the program; bit 27 clear indicates register *$27* is not used by the program.

`ri_cprmask` This member contains the bit-mask of co-processor registers used by the program. The MIPS RISC architecture supports up to four co-processors, each with 32 registers. Each array element corresponds to one set of co-processor registers. Each of the bits within the element corresponds to individual register in the co-processor register set. The 32 bits of the words correspond to the 32 registers, with bit number 31 corresponding to register 31, bit number 30 to register 30, etc. Set bits indicate the corresponding register is used by the program; clear bits indicate the program does not use the corresponding register.

`ri_gp_value` This member contains the gp register value. In relocatable object files it is used for relocation of the `R_MIPS_GPREL` and `R_MIPS_LITERAL` relocation types.

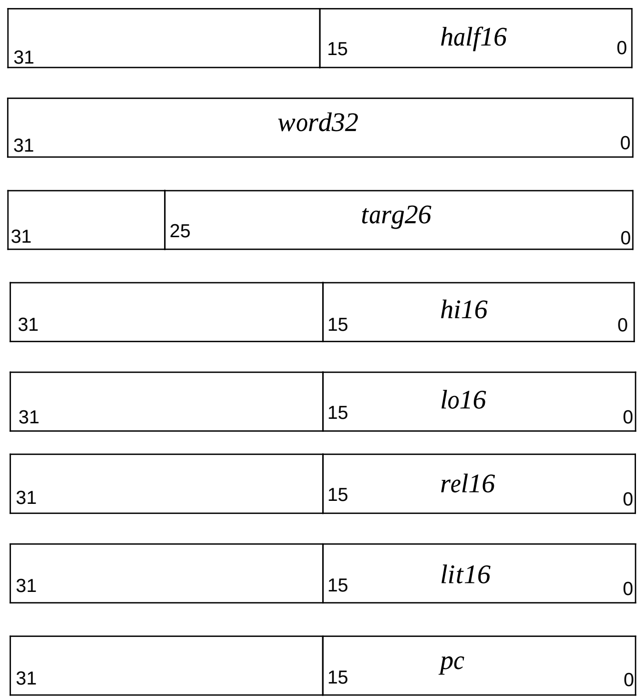
NOTE Only co–processor 1 can be used by *ABI*–compliant programs. This means that only the `ri_cprmask[1]` array element can have a non–zero value. `ri_cpr-mask[0]`, `ri_cprmask[2]`, and `ri_cprmask[3]` must all be zero in an *ABI*–compliant program.

Relocation

Relocation Types

Relocation entries describe how to alter the following instruction and data fields shown in Figure 4-10; bit numbers appear in the lower box corners.

Figure 4–10: Relocatable Fields



Calculations below assume the actions are transforming a relocatable file into either an executable or a shared object file. Conceptually, the linker merges one or more relocatable files to form the output. It first determines how to combine and locate the input files; then it updates the symbol values, and finally it 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 — Represents the addend used to compute the value of the relocatable field.

AHL — Identifies another type of addend used to compute the value of the relocatable field. See the note below for more detail.

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

S — Represents the value of the symbol whose index resides in the relocation entry, unless the the symbol is `STB_LOCAL` and is of type `STT_SECTION` in which case S represents the original `sh_addr` minus the final `sh_addr`.

G — Represents the offset into the global offset table at which the address of the relocation entry symbol resides during execution. See ‘‘Coding Examples’’ in Chapter 3 and ‘‘Global Offset Table’’ in Chapter 5 for more information.

GP — Represents the final `gp` value to be used for the relocatable, executable, or shared object file being produced.

GP0 — Represents the `gp` value used to create the relocatable object.

EA — Represents the effective address of the symbol prior to relocation.

L — Represents the `.lit4` or `.lit8` literal table offset. Prior to relocation the addend field of a literal reference contains the offset into the global data area. During relocation, each literal section from each contributing file is merged and sorted, after which duplicate entries are removed and the section compressed, leaving only unique entries. The relocation factor L is the mapping from the old offset of the original `gp` to the value of `gp` used in the final file.

A relocation entry `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. Because MIPS uses only `Elf32_Rel` relocation entries, the relocated field holds the addend.

The AHL addend is a composite computed from the addends of two consecutive relocation entries. Each relocation type of `R_MIPS_HI16` must have an associated `R_MIPS_LO16` entry immediately following it in the list of relocations.

These relocation entries are always processed as a pair and both addend fields contribute to the AHL addend. If `AHI` and `ALO` are the addends from the paired `R_MIPS_HI16` and `R_MIPS_LO16` entries, then the addend `AHL` is computed as `(AHI << 16) + (short)ALO`. `R_MIPS_LO16` entries without an `R_MIPS_HI16` entry immediately preceding are orphaned and the previously defined `R_MIPS_HI16` is used for computing the addend.

NOTE The field names in Table 4–11 tell whether the relocation type checks for overflow. A calculated relocation value can be larger than the intended field, and a relocation type can verify (*V*) the value fits or truncate (T) the result. As an example, *V–half16* means the computed value cannot have significant non–zero bits outside the *half16* field.

Figure 4–11: Relocation Types

Name	Value	Field	Symbol	Calculation
R_MIPS_NONE	0	none	local	none
R_MIPS_16	1	V–half16	external	S + sign–extend(A)
	1	V–half16	local	S + sign–extend(A)
R_MIPS_32	2	T–word32	external	S + A
	2	T–word32	local	S + A
R_MIPS_REL32	3	T–word32	external	A – EA + S
R_MIPS_REL32	3	T–word32	local	A – EA + S
R_MIPS_26	4	T–targ26	local	(((A << 2) | \ (P & 0xf0000000) + S) >> 2
	4	T–targ26	external	(sign–extend(A < 2) + S) >> 2
R_MIPS_HI16	5	T–hi16	external	((AHL + S) – \ (short)(AHL + S)) >> 16
	5	T–hi16	local	((AHL + S) – \ (short)(AHL + S)) >> 16
	5	V–hi16	_gp_disp	(AHL + GP – P) – (short) \ (AHL + GP – P)) >> 16
R_MIPS_LO16	6	T–lo16	external	AHL + S
	6	T–lo16	local	AHL + S
	6	V–lo16	_gp_disp	AHL + GP – P + 4
R_MIPS_GPREL16	7	V–rel16	external	sign–extend(A) + S + GP
	7	V–rel16	local	sign–extend(A) + S + GP0 – GP
R_MIPS_LITERAL	8	V–lit16	local	sign–extend(A) + L
R_MIPS_GOT16	9	V–rel16	external	G
	9	V–rel16	local	see below
R_MIPS_PC16	10	V–pc16	external	sign–extend(A) + S – P
R_MIPS_CALL16	11	V–rel16	external	G
R_MIPS_GPREL32	12	T–word32	local	A + S + GP0 – GP
R_MIPS_GOTHI16	21	T-hi16	external	(G - (short)G) >> 16 + A
R_MIPS_GOTLO16	22	T-lo16	external	G & 0xffff
R_MIPS_CALLHI16	30	T-hi16	external	(G - (short)G) >> 16 + A
R_MIPS_CALLLO16	31	T-lo16	external	G & 0xffff

In the *Symbol* column in the table above, *local* refers to a symbol referenced by the symbol table index in the relocation entry STB_LOCAL/STT_SECTION. Otherwise, the relocation is considered an *external* relocation. See below for _gp_disp relocations.

The R_MIPS_REL32 relocation type is the only relocation performed by the dynamic linker. The value EA used by the dynamic linker to relocate an

R_MIPS_REL32 relocation depends on its r_symndx value. If the relocation entry r_symndx is less than DT_MIPS_GOTSYM, the value of EA is the symbol st_value plus displacement. Otherwise, the value of EA is the value in the GOT entry corresponding to the relocation entry r_symndx. The correspondence between the GOT and the dynamic symbol table is described in the "Global Offset Table" section in Chapter 5.

If an R_MIPS_GOT16 refers to a locally defined symbol, then the relocation is done differently than if it refers to an external symbol. In the local case, the R_MIPS_GOT16 must be followed immediately with a R_MIPS_LO16 relocation. The AHL addend is extracted and the section in which the referenced data item resides is determined (requiring that all sections in an object module have unique addresses and not overlap). From this address the final address of the data item is calculated. If necessary, a global offset table entry is created to hold the high 16 bits of this address (an existing entry is used when possible). The *rel16* field is replaced by the offset of this entry in the global offset table. The *lo16* field in the following R_MIPS_LO16 relocation is replaced by the low 16 bits of the actual destination address. This is meant for local data references in position-independent code so that only one global offset table entry is necessary for every 64 KBytes of local data.

The first instance of R_MIPS_GOT16, R_MIPS_CALL16, R_MIPS_GOT_HI16, R_MIPS_CALL_HI16, R_MIPS_GOT_LO16, or R_MIPS_CALL_LO16. Relocations cause the link editor to build a global offset table if one has not already been built.

The symbol name _gp_disp is reserved. Only R_MIPS_HI16 and R_MIPS_LO16 relocations are permitted with _gp_disp. These relocation entries must appear consecutively in the relocation section and they must reference consecutive relocation area addresses.

R_MIPS_CALL16, R_MIPS_CALL_HI16, and R_MIPS_CALL_LO16 relocation entries load function addresses from the global offset table and indicate that the dynamic linker can perform lazy binding. See "Global Offset Table" in Chapter 5.

Program Loading

As the system creates or augments a process image, it logically copies a file 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 a logical page during execution. 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 virtual addresses are zero, modulo the file system block size.

Virtual addresses and file offsets for MIPS segments are congruent modulo 64 KByte (0x10000) or larger powers of 2. Because 64 KBytes is the maximum page size, the files are suitable for paging regardless of physical page size.

Figure 5-1: Example Executable File

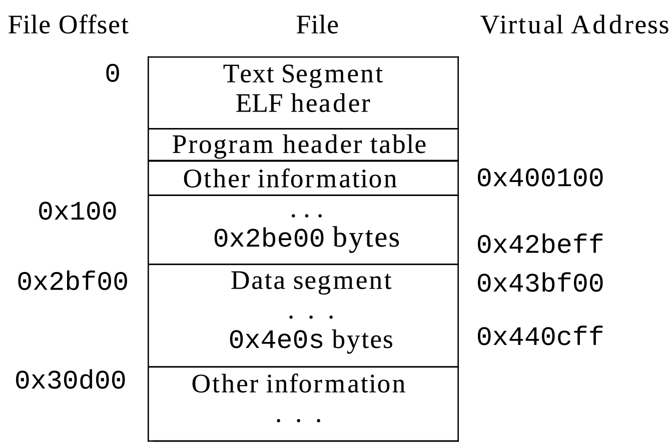


Figure 5-2: Program Header Segments

Member	Text	Data
p_type	PT_LOAD	PT_LOAD
p_offset	0	0x2bf00
p_vaddr	400100	0x43bf00
p_paddr	unspecified	unspecified
p_filesz	0x2bf00	0x4e00
p_memsz	0x2bf00	0x5e24
p_flags	PF_R+PF_X	PF_R+PF_W+PF_X
p_align	0x10000	0x10000

Because the page size can be larger than the alignment restriction of a segment file offset, up to four file pages can hold impure text or data (depending on page size and file system block size).

- The first text page contains the ELF header, the program header table, and other information.
- The last text page can hold a copy of the beginning of data.
- The first data page can have a copy of the end of text.
- The last data page can contain file information not relevant to the running process.

Logically, the system enforces the memory permissions as if each segment were complete and separate; segment addresses are adjusted to ensure each logical page in the address space has a single set of permissions. In the example in Figure 5-1, the file region holding the end of text and the beginning of data is mapped twice: once at one virtual address for text and once 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 the last data page of a file includes information not in the logical memory page, the extraneous data must be set to zero, rather than 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.

There is one aspect of segment loading that 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, with the system using the `p_vaddr` values unchanged as virtual addresses.

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

Figure 5-3: Example Shared Object Segment Addresses

Source	Text	Data	Base Address
File	0x200	0x2a400	0x0
Process 1	0x50000200	0x5002a400	0x50000000
Process 2	0x50010200	0x5003a400	0x50010000
Process 3	0x60020200	0x6004a400	0x60020000
Process 4	0x60030200	0x6005a400	0x60030000

NOTE In addition to maintaining the relative positionsof the segments, the system must also ensure that relocations occur in 64 KByte increments; position–independent code relies on this property.

NOTE By convention, no more than one segment will occupy addresses in the same chunk of memory, modulo 256 KBytes.

Program Header

There is one program header type specific to this supplement.

Figure 5-4: MIPS Specific Segment Types, `p_type`

Name	Value
PT_MIPS_REGINFO	0x70000000

`PT_MIPS_REGINFO` Specifies register usage information for the executable or shared object; it cannot occur more than once in a file. Its presence is mandatory and it must precede any loadable segment entry. It identifies one `.reginfo` type section. See Register Information" in Chapter 4 for more information.

Segment Contents

Figures 5-5 and 5-6 below illustrate typical segment contents for a MIPS executable or shared object. The actual order and membership of sections within a segment may alter the examples below.

Figure 5-5: Text Segment

.reginfo
.dynami
.liblist
.rel.dyn
.conflict
.dynstr
.dynsym
.hash
.rodata
.text

Figure 5-6: Data Segment

.got
.lit4
.lit8
.sdata
.data
.sbss
.bss

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.

Figure 5-7: Dynamic Array Tags `d_tag`

DT_MIPS_RLD_VERSION	0x70000001	d_val	mandatory	mandatory
DT_MIPS_TIME_STAMP	0x70000002	d_val	optional	optional
DT_MIPS_ICHECKSUM	0x70000003	d_val	optional	optional
DT_MIPS_IVERSION	0x70000004	d_val	optional	optional
DT_MIPS_FLAGS	0x70000005	d_val	mandatory	mandatory
DT_MIPS_BASE_ADDRESS	0x70000006	d_ptr	mandatory	mandatory
DT_MIPS_CONFLICT	0x70000008	d_ptr	optional	optional
DT_MIPS_LIBLIST	0x70000009	d_ptr	optional	optional
DT_MIPS_LOCAL_GOTNO	0x7000000A	d_val	mandatory	mandatory
DT_MIPS_CONFLICTNO	0x7000000B	d_val	optional	optional
DT_MIPS_LIBLISTNO	0x70000010	d_val	optional	optional
DT_MIPS_SYMTABNO	0x70000011	d_val	mandatory	mandatory
DT_MIPS_UNREFEXTNO	0x70000012	d_val	optional	optional
DT_MIPS_GOTSYM	0x70000013	d_val	mandatory	mandatory
DT_MIPS_HIPAGENO	0x70000014	d_val	optional	optional
DT_MIPS_RLD_MAP	0x70000016	d_ptr	mandatory	ignored
DT_PLTGOT	3	d_ptr	mandatory	mandatory
DT_RPATH	15	d_val	optional	optional

`DT_MIPS_RLD_VERSION`

This element holds a 32-bit version id for the *Runtime Linker Interface*. This will start at integer value 1.

`DT_MIPS_TIME_STAMP`

This element holds a 32-bit time stamp.

`DT_MIPS_ICHECKSUM`

This element holds the sum of all external strings and common sizes.

DT_MIPS_IVERSION
: This element holds an index into the object file string table. The version string is a series of version strings separated by colons (:). An index value of zero means no version string was specified.

DT_MIPS_FLAGS
: This element holds a set of 1-bit flags. Flag definitions appear below.

DT_MIPS_BASE_ADDRESS
: This member holds the *base address* of the segment. That is, it holds the virtual address of the segment as if the the segment were actually loaded at the addressed specified at static link time. It can be adjusted when the operating system kernel actually maps segments. It is used to adjust pointers based on the difference between the static link time value and the actual address.

DT_MIPS_CONFLICT
: This member holds the address of the `.conflict` section.

DT_MIPS_LIBLIST
: This member holds address of the `.liblist` section.

DT_MIPS_LOCAL_GOTNO
: This member holds the number of local global offset table entries.

DT_MIPS_CONFLICTNO
: This member holds the number of entries in the `.conflict` section. This field is mandatory if there is a `.conflict` section.

DT_PLTGOT
: This member holds the address of the `.got` section.

DT_MIPS_SYMTABNO
: This member holds the number of entries in the `.dynsym` section.

DT_MIPS_LIBLISTNO
: This member holds the number of entries in the `.liblist` section.

DT_MIPS_UNREFEXTNO
: This member holds the index into the dynamic symbol table which is the entry of the first external symbol that is not referenced within the same object.

DT_MIPS_GOTSYM — This member holds the index of the first dynamic symbol table entry that corresponds to an entry in the global offset table. See "Global Offset Table" in this chapter.

DT_MIPS_HIPAGENO — This member holds the number of page table entries in the global offset table. A page table entry here refers to a 64 Kb chunk of data space. This member is used by profiling tools and is optional.

DT_RPATH — This member optionally appears in a shared object. If it is present in a shared object at static link time, it is propagated to the final executable's DT_RPATH.

DT_DEBUG — This member is specifically disallowed.

DT_MIPS_RLD_MAP — This member is used by debugging. It contains the address of a 32-bit word in the .data section which is supplied by the compilation environment. The word's contents are not specified and programs using this value are not *ABI* - compliant.

Figure 5-8: Dynamic section, DT_MIPS_FLAGS

Name	Value	Meaning
RHF_NONE	0x00000000	none
RHF_QUICKSTART	0x00000001	use shortcut pointers
RHF_NOTPOT	0x00000002	hash size not power of two
RHF_NO_LIBRARY_REPLACEMENT	0x00000004	ignore LD_LIBRARY_PATH

The RHF_NO_LIBRARY_REPLACEMENT flag directs the dynamic linker to ignore the LD_LIBRARY_PATH environment variable when searching for shared objects.

Shared Object Dependencies

The *System V ABI* defines the default library search path to be /usr/lib; MIPS defines the default library search path to be /lib:/usr/lib:/usr/lib/cmplrs/cc.

Global Offset Table

In general, position-independent code cannot contain absolute virtual addresses.

Global offset tables (or GOTs) hold absolute addresses in private data, making the addresses available without compromising position-independence and sharability of a program text. A program references its global offset table using position-independent addressing and extracts absolute values, thus redirecting position-independent references to absolute locations.

The global offset table is split into two logically separate subtables: locals and externals. Local entries reside in the first part of the global offset table. The value of the dynamic tag `DT_MIPS_LOCAL_GOTNO` holds the number of local global offset table entries. These entries only require relocation if they occur in a shared object and the shared object memory load address differs from the virtual address of the loadable segments of the shared object. As with defined external entries in the global offset table, these local entries contain actual addresses.

External entries reside in the second part of the global offset table. Each entry in the external section corresponds to an entry in the global offset table mapped part of the `.dynsym` section (see "Symbols" below for a definition). The first symbol in the `.dynsym` section corresponds to the first word of the global offset table; the second symbol corresponds to the second word, and so on. Each word in the external entry part of the global offset table contains the *actual address* for its corresponding symbol. The external entries for defined symbols must contain actual addresses. If an entry corresponds to an undefined symbol and the global offset table entry contains a zero, the entry must be resolved by the dynamic linker, even if the dynamic linker is performing a *quickstart*. See "Quickstart" below for more information.

The following table details the various possibilities for the initial state of the global offset table mapped dynamic symbol table section and the global part of the global offset table.

Figure 5-9: Initial State, global GOT and .dynsym

Section	Type	st_value	GOT Entry	Comments
SHN_UNDEF	STT_FUNC	0	0/QS	1
SHN_UNDEF	STT_FUNC	stub addr	stub address/ QS	2
SHN_UNDEF SHN_COMMON	any	0/alignment	0/QS	
all others	STT_FUNC	address	stub address/ address	2
all others	any	address	address	3

QS stands for the *Quickstart* value of the symbol.

Comments:
1: had relocations related to taking the function's address
2: only had call related relocations defined STT_FUNC
3: non-STT_FUNC defined globals

After the system creates memory segments for a loadable object file, the dynamic linker can process the relocation entries. The only relocation entries remaining are type R_MIPS_REL32 referring to data containing addresses. The dynamic linker determines the associated symbol (or section) values, calculates their absolute addresses, and sets the proper values. Although the absolute addresses may be unknown when the link editor builds an object file, the dynamic linker knows the addresses of all memory segments and can find the correct symbols, thus calculating the absolute addresses contained therein.

The dynamic linker relocates the global offset table by first adding the difference between the base where the shared object is loaded and the value of the dynamic tag DT_MIPS_BASE_ADDRESS to all local global offset table entries. Next, the global GOT entries are relocated. For each global GOT entry the following relocation is performed:

Figure 5-10: Global Offset Table Relocation Algorithm

Section	Type	st_value	GOT Entry	Relocation
SHN_UNDEF	STT_FUNC	0	0 / QS	1
SHN_UNDEF	STT_FUNC	stub addr	stub addr	2
SHN_UNDEF	STT_FUNC	stub addr	!= stub addr	3
SHN_UNDEF SHN_COMMON	all others	any	0 / QS	1
all others	STT_FUNC	address	stub address != address*	2
all others	all others	address	address	1

* Stub address must be in this executable and can only be applied the first time the GOT is modified.

Relocation:
1: resolve immediately or use *Quickstart* value
2: add run-time displacement to GOT entry
3: set GOT entry to stub address plus run-time displacement

Certain optimizations are possible with information from Quickstart. An ABI-compliant system performing such optimizations guarantees that the values of the GOT entries are the same as if the dynamic linker performed the relocation algorithm described in Figure 5-10.

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

The zero entry in the global offset table is reserved to hold the address of the entry point in the dynamic linker to call when lazy resolving text symbols. The dynamic linker must always initialize this entry regardless of whether lazy binding is or is not enabled.

The system can choose different memory segment addresses for the same shared object in different programs; it can 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.

Calling Position–Independent Functions

The global offset table is used to hold addresses of position-independent functions as well as data addresses. It is not possible to resolve function calls from one executable file or shared object to another at static link time, so all of the function address entries in the global offset table are normally resolved at execution time. The dynamic linker then resolves all of these undefined relocation entries at run-time. Through the use of specially constructed pieces of code known as stubs, this run-time resolution can be be deferred through a technique known as " binding, lazy binding".

Using this technique, the link editor (or a combination of the compiler, assembler, and link editor) builds a stub for each called function, and allocates a global offset table entry that initially points to the stub. Because of the normal calling sequence for position-independent code, the call ends up invoking the stub the first time the call is made.

Figure 5-11: Sample Stub Code

```
stub_xyz: .
   lw      t9, 0(gp)
   move    t7, ra
   jal     t9
   li      t8, .dynsym_index    # branch delay slot
```

In the example in Figure 5-11, the stub code loads register `t9` with an entry from the global offset table which contains a well-known entry point in the dynamic linker; it also loads register `t8` with the index into the `.dynsym` section of the referenced external. The code must save register `ra` in register `t7` and transfer control to the dynamic linker.

The dynamic linker determines the correct address for the actual called function and replaces the address of the stub in the global offset table with the address of the function.

Most undefined text references can be handled by lazy text evaluation except when the address of a function is relocated using relocations of type
`R_MIPS_CALL16 or R_MIPS_26.`

The LD_BIND_NOW environment variable can also change dynamic linking behavior. If its value is non-null, the dynamic linker evaluates all symbol table entries of type STT_FUNC, replacing their stub addresses in the global offset table with the actual address of the referenced function.

NOTE Lazy binding generally improves overall application performance because unused symbols do not incur the dynamic linking overhead. Nevertheless, two situations make lazy binding undesirable for some applications. First, the initial reference to a shared object function takes longer than subsequent calls, because the dynamic linker intercepts the call to resolve the symbol. Some applications cannot tolerate this unpredictability. Second, if an error occurs and the dynamic linker cannot resolve the symbol, the dynamic linker terminates 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.

Symbols

All externally visible symbols, both defined and undefined, must be hashed into the hash table.

Undefined symbols of type STT_FUNC, which have been referenced only by R_MIPS_CALL16 and R_MIPS_26 relocations, can contain non-zero values in the their st_value field, denoting the stub address used for lazy evaluation for this symbol. The run-time linker uses this to reset the global offset table entry for this external to its stub address when unlinking a shared object. All other undefined symbols must contain zero in their st_value fields.

The dynamic symbol table, like all ELF symbol tables, is divided into local and global parts. The global part of the dynamic symbol table is further divided into two parts: symbols that do not have GOT entries associated with them and symbols that do have GOT entries associated with them. The part of the dynamic symbol table with GOT entries is called the "global offset table mapped" part or "GOT mapped" part. Symbols with GOT entries have a one-to-one mapping with the global part of the GOT.

The value of the dynamic tag DT_MIPS_GOTSYM is the index of the first symbol with a global offset table entry in the dynamic symbol table.

Relocations

There may be only one dynamic relocation section to resolve addresses in data. It must be called .rel.dyn. Executables can contain normal relocation sections in addition to a dynamic relocation section. The normal relocation sections may contain resolutions for any absolute values in the main program. The dynamic linker does not resolve these or relocate the main program.

As noted previously, only `R_MIPS_REL32` relocation entries are supported in the dynamic relocation section.

Because sufficient information is available in the `.dynamic` section, the GOT has no relocation information. The relocation algorithm for the GOT is described above.

The entries in the dynamic relocation section must be ordered by increasing `r_symndx` value.

Ordering

To take advantage of *Quickstart* functionality, the `.dynsym and .rel.dyn` sections must obey ordering constraints. The GOT-mapped portion of the `.dynsym` section must be ordered on increasing values in the `st_value` field. This requires that the `.got` section have the same order, since it must correspond to the `.dynsym` section.

The `.rel.dyn` section must have all local entries first, followed by the external entries. Within these sub-sections, the entries must be ordered by symbol index.

Quickstart

The MIPS supplement to the *ABI* defines two sections which are useful for faster start-up of programs when the programs have been linked with dynamic shared objects. The group of structures defined in these sections allow the dynamic linker to operate more efficiently than when these sections are not present. These additional sections are also used for more complete dynamic shared object version control.

NOTE: An *ABI* compliant system can ignore the sections defined here, but if it supports one of these sections, it must support both of them. If you relink or relocate the object file on secondary storage and cannot process these sections, you must delete them.

Shared Object List

A shared object list section is an array of structures that contain information about the various dynamic shared objects used to statically link this object file. Each separate shared object used generates one `Elf32_Lib` array element. The shared object list is used for more complete shared object version control.

Figure 5-12: Shared Object Information Structure

```
typedef struct {
   Elf32_Word     l_name;
   Elf32_Word     l_time_stamp;
   Elf32_Word     l_checksum;
   Elf32_Word     l_version;
   Elf32_Word     l_flags;
} Elf32_Lib;
```

`l_name` This member specifies the name of a shared object. Its value is a string table index. This name can be a trailing component of the path to be used with `RPATH + LD_LIBPATH` or a name containing '/'s, which is relative to '.', or it can be a full pathname.

`l_time_stamp` This member's value is a 32 bit time stamp. It can be combined with the l_checksum value and the l_version string to form an unique id for this shared object.

l_checksum This member's value is the sum of all externally visible symbol's string names and common sizes.

l_version This member specifies the interface version. Its value is a string table index. The interface version is a single string containing no colons (:). It is compared against a colon separated string of versions pointed to by a dynamic section entry of the shared object. Shared objects with matching names are considered incompatible if the interface version strings are deemed incompatible. An index value of zero means no version string is specified.

flags This is a set of 1 bit flags. Flag definitions appear below.

Figure 5-13: Library Flags, l_flags

Name	Value	Meaning
LL_EXACT_MATCH	0x00000001	require exact match
LL_IGNORE_INT_VER	0x00000002	ignore interface version

LL_EXACT_MATCH At run-time use a unique id composed of the l_time_stamp, l_checksum, and l_version fields to demand that the run-time dynamic shared library match exactly the shared library used at static link time.

LL_IGNORE_INT_VER

At run-time, ignore any version incompatibilities between the dynamic shared library and the library used at static link time.

Normally, if neither LL_EXACT_MATCH nor LL_IGNORE_INT_VER bits are set, the dynamic linker requires that the version of the dynamic shared library match at least one of the colon separated version strings indexed by the l_version string table index.

Conflict Section

The `.conflict` section is an array of indexes into the `.dynsym` section. Each index identifies a symbol whose attributes conflict with a shared object on which it depends, either in type or size such that this definition will preempt the shared object's definition. The dependent shared object is identified at static link time.

Figure 5-14: Conflict Section

```
typedef Elf32_Addr Elf32_Conflict;
```

System Library

Additional Entry Points

The following routines are included in the **libsys** library to provide entry points for the required source-level interfaces listed in the *System V ABI*. A description and syntax summary for each function follows the table.

Figure 6-1: `libsys` **Additional Required Entry Points**

`_fxstat`	`_lxstat`	`_xmknod`	`_xstat`	`nuname`	`_nuname`

int _ fxstat (int, int, struct stat *);

The semantics of this function are identical to those of the `fstat` (BA_OS) function described in the *System V Interface Definition,Third Edition*. Its only difference is that it requires an extra first argument whose value must be 2.

int _lxstat (int, char *, struct stat *);

The semantics of this function are identical to those of the `lstat` (BA__OS) function described in the *System V Interface Definition*, *Third Edition*. Its only difference is that it requires an extra first argument whose value must be 2.

int _nuname (struct utsname *);

The semantics and syntax of this function are identical to those of the `uname`(BA__OS) function described in the *System V Interface Definition*,-*Third Edition*. The symbol _nuname is also available with the same semantics.

int _xmknod(int, char *, mode_t, dev_t);

The semantics and syntax of this function are identical to those of the `mknod`(BA__OS) function described in the *System V Interface Definition*,-*Third Edition*. Its only difference is that it requires an extra first argument whose value must be 2.

int _xstat(int, char *, struct stat *);

The semantics of this function are identical to those of the `stat`(BA_OS) function described in the *System V Interface Definition*, *Third Edition*. Its only difference is that it requires an extra first argument whose value must be 2.

Support Routines

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

Figure 6-2: `libsys` **Support Routines**

`sbrk` `_sbrk` `_sqrt_s` `_sqrt_d` `_test_and_set` `_flush_cache`

The routines listed below employ the standard calling sequence described in Chapter 3, "Function Calling Sequence."

char *sbrk(int incr);

This function adds *incr* bytes to the *`break value`* and changes the allocated space accordingly. *`Incr`* can be negative, in which case the amount of allocated space is decreased. The break value is the address of the first allocation beyond the end of the data segment. The amount of allocated space increases as the break value increases. Newly allocated space is set to zero. If, however, the same memory is reallocated to the same process, its contents are undefined. Upon successful completion, `sbrk` returns the old break value. Otherwise, it returns -1 and sets `errno` to indicate the error. The symbol `_sbrk` is also available with the same semantics. NOTE: mixing sbrk & malloc is hazardous to your program's health.

`float _sqrt_s(float v)`

This function computes $\sqrt{v}$ using single-precision floating point arithmetic and returns the resulting value. The result is rounded as if calculated to infinite precision and then rounded to single-precision according to the current rounding modes specified by the floating point control/status register. If the value is -0, the result is -0. `_sqrt_s` can trigger the floating point exceptions *Invalid Operation* when v is less than 0 or *Inexact*.

`double _sqrt_d(double v)`

This function computes $\sqrt{v}$ using double-precision floating point arithmetic and returns the resulting value. The result is rounded as if calculated to infinite precision and then rounded to double-precision according to the current rounding modes specified by the floating point control / status register. If the value is -0, the result is -0. `_sqrt_d` can trigger the floating point exceptions *Invalid Operation* when v is less than 0 or *Inexact*.

`int _test_and_set(int *p, int v)`

This function performs an atomic *test and set* operation on the integer pointed to by `p`. It effectively performs the following operations, but with a guarantee that no other process executing on the system can interrupt the operation.

```
temp = *p;
*p = v;
return(temp);
```

`int _flush_cache(char *addr, int nbytes, int cache)`

This function flushes the contents of the associated cache(s) for user program addresses in the range `addr` to `addr + nbytes – 1`. `Cache` can be:

`ICACHE` - Flush only the instruction cache.

`DCACHE` - Flush only the data cache.

`BCACHE` - Flush both instruction and data cache.

These definitions are in the include file <`sys/cachectl.h`>. The function returns zero when no errors are detected and returns -1 otherwise, with the error cause indicated in `errno`. On error, the two possible `errno` values are either `EINVAL`, indicating an invalid value for the `cache` parameter, or `EFAULT`, indicating some part or all of the address range specified is not accessable.

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 MIPS processor architecture. Declarations for the data objects listed below can be found in the "Data Definitions" section.

Figure 6-3: `libsys`**, Global External Data Symbols**

`__huge_val`

Application Constraints

As described above, ***libsys*** provides symbols for applications. In a few cases, however, an application must 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 MIPS 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 the dynamic allocation area of a program, called the *heap*. Typically, the heap begins immediately after the data segment of the program executable file.

`extern _gp;` This symbol is defined by the link editor and provides the value used for the gp register for this executable or shared object 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 preserves the semantics of older (pre-ANSI) C, where conflicts exist with ANSI. Otherwise, the value is non-zero, and the program requires ANSI C semantics.

`extern _DYNAMIC_LINKING;`
This variable is a flag that the static linker sets to non-zero if the object is dynamically linked and is capable of linking with other dynamic shared objects at run time. The value is set to zero otherwise.

System Data Interfaces

Data Definitions

This section contains standard header files that describe system data. These header files are referred to by their names in angle brackets: <*name.h*> and <sys/*name.h*>. Included in these header files are macro 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 runs. While an *ABI*-conforming system must provide these interfaces, it is not required to contain the actual header files referenced here.

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

Figure 6-4: <assert.h>

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

Figure 6-5: <sys/cachectl.h>

```
#define ICACHE   0x1
#define DCACHE   0x2
#define BCACHE   (ICACHE | DCACHE)
```

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

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

extern unsigned char  __ctype[];

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

Figure 6-7: `<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];
};
```

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

```
extern int errno;

#define EPERM          1
#define ENOENT         2
#define ESRCH          3
#define EINTR          4
#define EIO            5
#define ENXIO          6
#define E2BIG          7
#define ENOEXEC        8
#define EBADF          9
#define ECHILD         10
#define EAGAIN         11
#define ENOMEM         12
#define EACCES         13
#define EFAULT         14
#define ENOTBLK        15
#define EBUSY          16
#define EEXIST         17
#define EXDEV          18
#define ENODEV         19
#define ENOTDIR        20
#define EISDIR         21
#define EINVAL         22
#define ENFILE         23
#define EMFILE         24
#define ENOTTY         25
#define ETXTBSY        26
#define EFBIG          27
#define ENOSPC         28
#define ESPIPE         29
```

Figure 6-8: `<errno.h>` (continued)

```
#define EROFS          30
#define EMLINK         31
#define EPIPE          32
#define EDOM           33
#define ERANGE         34
#define ENOMSG         35
#define EIDRM          36
#define ECHRNG         37
#define EL2NSYNC       38
#define EL3HLT         39
#define EL3RST         40
#define ELNRNG         41
#define EUNATCH        42
#define ENOCSI         43
#define EL2HLT         44
#define EDEADLK        45
#define ENOLCK         46
#define ENOSTR         60
#define ENODATA        61
#define ETIME          62
#define ENOSR          63
#define ENONET         64
#define ENOPKG         65
#define EREMOTE        66
#define ENOLINK        67
#define EADV           68
#define ESRMNT         69
```

Figure 6-8: `<errno.h>` (continued)

```
#define ECOMM           70
#define EPROTO          71
#define EMULTIHOP       74
#define EBADMSG         77
#define ENAMETOOLONG    78
#define EOVERFLOW       79
#define ENOTUNIQ        80
#define EBADFD          81
#define EREMCHG         82
#define ENOSYS          89
#define ELOOP           90
#define ERESTART        91
#define ESTRPIPE        92
#define ENOTEMPTY       93
#define EUSERS          94
#define ECONNABORTED    130
#define ECONNRESET      131
#define ECONNREFUSED    146
#define ESTALE          151
```

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

```
#define O_RDONLY        0
#define O_WRONLY        1
#define O_RDWR          2
#define O_APPEND        0x08
#define O_SYNC          0x10
#define O_NONBLOCK      0x80
#define O_CREAT         0x100
#define O_TRUNC         0x200
#define O_EXCL          0x400
#define O_NOCTTY        0x800

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

#define FD_CLOEXEC      1
#define O_ACCMODE       3

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

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

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

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

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

```
#define MM_NULL         0L

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

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

#define MM_NULLLBL      ((char *) NULL)
#define MM_NULLSEV      MM_NOSEV
#define MM_NULLMC       MM_NULL
#define MM_NULLTXT      ((char *) NULL)
#define MM_NULLACT      ((char *) NULL)
#define MM_NULLTAG      ((char *) NULL)

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

Figure 6-12: <ftw.h>

```
#define FTW_PHYS                   01
#define FTW_MOUNT                  02
#define FTW_CHDIR                  04
#define FTW_DEPTH                0  10

#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 6-13: <grp.h>

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

Figure 6-14: `<sys/ipc.h>`

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

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

#define IPC_PRIVATE      (key_t)0

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

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

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

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

#define MON_1            15
#define MON_2            16
#define MON_3            17
#define MON_4            18
#define MON_5            19
#define MON_6            20
#define MON_7            21
#define MON_8            22
#define MON_9            23
#define MON_10           24
#define MON_11           25
#define MON_12           26
```

Figure 6-15: **<**`langinfo.h`**>** (continued)

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

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

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

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

```
#define MB_LEN_MAX      5

#define ARG_MAX         *
#define CHILD_MAX       *
#define MAX_CANON       *
#define NGROUPS_MAX     *
#define LINK_MAX        *
#define NAME_MAX        *
#define OPEN_MAX        *
#define PASS_MAX        *
#define PATH_MAX        *
#define PIPE_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        2147483647
```

Figure 6-17: `<locale.h>`

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

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

Figure 6-18: `<math.h>`

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

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

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

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

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

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

Figure 6-20: `<sys/mount.h>`

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

Figure 6-21: `<sys/msg.h>`

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

#define MSG_NOERROR         010000
```

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

```
struct  netconfig{
     char                *nc_netid;
     unsigned long       nc_semantics;
     unsigned long       nc_flag;
     char                *nc_protofmly;
     char                *nc_proto;
     char                *nc_device;
     unsigned long       nc_nlookups;
     char                **nc_lookups;
     unsigned long       nc_unused[8];
};

#define NC_TPI_CLTS      1
#define NC_TPI_COTS      2
#define NC_TPI_COTS_ORD  3
#define NC_TPI_RAW       4
#define NC_NOFLAG        00
#define NC_VISIBLE       01
```

Figure 6-22: `<netconfig.h>` (continued)

```
#define NC_NOPROTOFMLY  ”–”
#define NC_LOOPBACK     ”loopback”
#define NC_INET         ”inet”
#define NC_IMPLINK      ”implink”
#define NC_PUP          ”pup”
#define NC_CHAOS        ”chaos”
#define NC_NS           ”ns”
#define NC_NBS          ”nbs”
#define NC_ECMA         ”ecma”
#define NC_DATAKIT      ”datakit”
#define NC_CCITT        ”ccitt”
#define NC_SNA          ”sna”
#define NC_DECNET       ”decnet”
#define NC_DLI          ”dli”
#define NC_LAT          ”lat”
#define NC_HYLINK       ”hylink”
#define NC_APPLETALK    ”appletalk”
#define NC_NIT          ”nit”
#define NC_IEEE802      ”ieee802”
#define NC_OSI          ”osi”
#define NC_X25          ”x25”
#define NC_OSINET       ”osinet”
#define NC_GOSIP        ”gosip”
#define NC_NOPROTO      ”–”
#define NC_TCP          ”tcp”
#define NC_UDP          ”udp”
#define NC_ICMP         ”icmp”
```

Figure 6-23: `<netdir.h>`

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

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

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

#define ND_BADARG          –2
#define ND_NOMEM           –1
#define ND_OK              0
#define ND_NOHOST          1
#define ND_NOSERV          2
#define ND_NOSYM           3
#define ND_OPEN            4
#define ND_ACCESS          5
#define ND_UKNWN           6
#define ND_NOCTRL          7
#define ND_FAILCTRL        8
#define ND_SYSTEM          9
```

Figure 6-23: `<netdir.h>` (continued)

```
#define ND_HOSTSERV             0
#define ND_HOSTSERVLIST         1
#define ND_ADDR                 2
#define ND_ADDRLIST             3

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

#define ND_SET_BROADCAST        1
#define ND_SET_RESERVEDPORT     2
#define ND_CHECK_RESERVEDPORT   3
#define ND_MERGEADDR            4
```

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

```
#define NL_SETD         1

typedef int nl_item ;
typedef void *nl_catd;
```

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

```
#define CANBSIZ         256
#define HZ              100

#define NGROUPS_UMIN    0

#define MAXPATHLEN      1024
#define MAXSYMLINKS     30
#define MAXNAMELEN      256

#define NADDR           13

#define NBBY            8
#define NBPSCTR         512
```

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

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

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

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

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

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

typedef long id_t;

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

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

Figure 6-27: `<sys/procset.h>` (continued)

```
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 6-28: `<pwd.h>`

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

Figure 6-29: `<sys/resource.h>`

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

typedef unsigned long rlim_t;

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

Figure 6-30: `<rpc.h>`

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

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

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

Figure 6-30: `<rpc.h>` (continued)

```
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;

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

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

Figure 6-30: `<rpc.h>` (continued)

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

#define RPC_PMAPFAILURE RPC_RPCBFAILURE
```

Figure 6-30: `<rpc.h>` (continued)

```
#define RPC_AYSOCK  -1
#define RPC_ANYFD   RPC_ANYSOCK

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

Figure 6-30: `<rpc.h>` (continued)

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

#define FEEDBACK_REXMIT1        1
#define FEEDBACK_OK             2

#define CLSET_TIMEOUT           1
#define CLGET_TIMEOUT           2
#define CLGET_SERVER_ADDR       3
#define CLGET_FD                6
#define CLGET_SVC_ADDR          7
#define CLSET_FD_CLOSE          8
#define CLSET_FD_NCLOSE         9
#define CLSET_RETRY_TIMEOUT     4
#define CLGET_RETRY_TIMEOUT     5
```

Figure 6-30: `<rpc.h>` (continued)

```
extern struct
rpc_createerr rpc_createerr;
enum xprt_stat{
      XPRT_DIED,
      XPRT_MOREREQS,
      XPRT_IDLE
};

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

Figure 6-30: `<rpc.h>` (continued)

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

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

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

 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
};
```

Figure 6-30: `<rpc.h>` (continued)

```
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;
};
```

Figure 6-30: <rpc.h> (continued)

```
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_rpcvers;
      unsigned long cb_prog;
      unsigned long cb_vers;
      unsigned long cb_proc;
      struct opaque_auth cb_cred;
      struct opaque_auth cb_verf;
};

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

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

Figure 6-30: `<rpc.h>` (continued)

```
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;
};

 struct authdes_cred{
      enum authdes_namekind adc_namekind;
      struct authdes_fullname adc_fullname;
      unsigned long adc_nickname;
};
```

Figure 6-30: `<rpc.h>` (continued)

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

typedef int (*xdrproc_t)()
#define NULL_xdrproc_t ((xdrproc_t)0)
```

Figure 6-30: `<rpc.h>` (continued)

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

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

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

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

```
#define SEM_UNDO    010000

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

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

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

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

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

```
#define _JBLEN          28
#define _SIGJBLEN       128

typedef int jmp_buf[_JBLEN];
typedef int sigjmp_buf[_SIGJBLEN];
```

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

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

#define SHM_RDONLY       010000
#define SHM_RND          020000
```

Figure 6-35: `<signal.h>`

```
#define SIGHUP        1
#define SIGINT        2
#define SIGQUIT       3
#define SIGILL        4
#define SIGTRAP       5
#define SIGABRT       6
#define SIGEMT        7
#define SIGFPE        8
#define SIGKILL       9
#define SIGBUS        10
#define SIGSEGV       11
#define SIGSYS        12
#define SIGPIPE       13
#define SIGALRM       14
#define SIGTERM       15
#define SIGUSR1       16
#define SIGUSR2       17
#define SIGCHLD       18
#define SIGPWR        19
#define SIGWINCH      20
#define SIGURG        21
#define SIGPOLL       22
#define SIGSTOP       23
#define SIGTSTP       24
#define SIGCONT       25
#define SIGTTIN       26
#define SIGTTOU       27
#define SIGXCPU       30
#define SIGXFSZ       31
```

Figure 6-35: `<signal.h` (continued)

```
#define SIG_BLOCK        1
#define SIG_UNBLOCK      2
#define SIG_SETMASK      3
#define SIG_ERR          (void(*)())-1
#define SIG_IGN          (void(*)())1
#define SIG_HOLD         (void(*)())2
#define SIG_DFL          (void(*)())0

#define SS_ONSTACK       0x00000001
#define SS_DISABLE       0x00000002
 struct sigaltstack {
         char            *ss_sp;
         int             ss_size;
         int             ss_flags;
};
typedef struct sigaltstackstack_t;
typedef struct { unsigned long sigbits[4];} sigset_t;

struct sigaction{
         int             sa_flags;
         void            (*sa_handler)();
         sigset_t        sa_mask;
         int             sa_resv[2];
};

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

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

```
#define ILL_ILLOPC      1
#define ILL_ILLOPN      2
#define ILL_ILLADR      3
#define ILL_ILLTRP      4
#define ILL_PRVOPC      5
#define ILL_PRVREG      6
#define ILL_COPROC      7
#define ILL_BADSTK      8
```

Figure 6-36: `<sys/siginfo.h>` (continued)

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

Figure 6-36: `<sys/siginfo.h>` (continued)

```
typedef struct siginfo{
      int  si_signo;
      int  si_code;
      int  si_errno;
      union {
        int _pad[SI_PAD];
        struct {
            pid_t  _pid;
            union {
            struct { uid_t _uid;} _kill;
            struct {
                  clock_t _utime;
                  int     _status;
                  clock_t _stime;
            } _cld;
        } _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_stime  _data._proc._pdata._cld._stime
#define si_utime  _data._proc._pdata._cld._utime
#define si_status _data._proc._pdata._cld._status
#define si_band   _data._file._band
#define si_fd     _data._file._fd
```

Figure 6-37: `<sys/stat.h>`

```
#define _ST_FTYPSZ  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;
         timestruc_t st_atim;
         timestruc_t st_mtim;
         timestruc_t st_ctim;
         long       st_blksize;
         long       st_blocks;
         char       st_fstype[_ST_FSTYPSZ];
         long       st_pad4[8];
};

#define st_atime    st_atim.tv_sec
#define st_mtime    st_mtim.tv_sec
#define st_ctime    st_ctim.tv_sec
```

Figure 6-37: `<sys/stat.h>` (continued)

```
#define S_IFMT           0xF000
#define S_IFIFO          0x1000
#define S_IFCHR          0x2000
#define S_IFDIR          0x4000
#define S_IFBLK          0x6000
#define S_IFREG          0x8000
#define S_IFLNK          0xA000
#define S_ISUID          04000
#define S_ISGID          02000
#define S_ISVTX          01000
#define S_IRWXU          00700
#define S_IRUSR          00400
#define S_IWUSR          00200
#define S_IXUSR          00100
#define S_IRWXG          00070
#define S_IRGRP          00040
#define S_IWGRP          00020
#define S_IXGRP          00010
#define S_IRWXO          00007
#define S_IROTH          00004
#define S_IWOTH          00002
#define S_IXOTH          00001

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

Figure 6-38: `<sys/statvfs.h>`

```
#define FSTYPSZ     16

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

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

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

```
typedef void *va_list;
#define va_end(list)       (void)0

#define va_start(list, name)\
   (void) (list = (void *)((char *) &. . . ))
#define va_arg(list, mode)\
   ((mode *)(list = (char *) ((((int)list +\
   (__builtin_alignof(mode)<=4?3:7)) &\
   (__builtin_alignof(mode)<=4?-4:-8))+sizeof(mode))))[-1]
```

NOTE The construction &... is a syntactic extension to ANSI C and may not be supported by all C compilers. The intended semantics are to set list to the address on the stack of the first incoming argument in the variable part of the argument list. See "Function Calling Sequence" in Chapter 3.

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

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

‡ The _file member of the FILE struct is moved to Level 2 as of Jan. 1, 1993.

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

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

#define _NFILE            100
#define NULL              0
#define BUFSIZ            4096
#define _IOFBF            0000
#define _IOLBF            0100
#define _IONBF            0004
#define _IOEOF            0020
#define _IOERR            0040
#define EOF               (-1)
#define FOPEN_MAX         60
#define FILENAME_MAX      1024

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

#define clearerr(p)       ((void)((p)->_flag &= ~(_IOERR|_I-
OEOF)))†
#define feof(p)           ((p)->_flag & _IOEOF)
#define ferror(p)         ((p)->_flag & _IOERR)†
#define fileno(p)         (p)->_file
#define L_ctermid         9
#define L_cuserid         9
#define P_tmpdir          "/var/tmp/"
```

† These macro definitions are moved to Level 2 as of Jan. 1, 1993.

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

```
typedef struct {
      int              _cnt;
      unsigned char    *_ptr;
      unsigned char    *_base;
      unsigned char    _flag;
      unsigned char    _file;‡
} FILE;

extern FILE            __iob[_NFILE];
```

NOTE

The macros `clearerr`, and `fileno` will be removed as a source interface in a future release supporting multi-processing. Applications should transition to the function equivalents of these macros in libc. Binary portability will be supported for existing applications.

CAUTIO

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 6-42: `<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 6-43: `<stropts.h>`

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

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

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

Figure 6-43: `<stropts.h>` (continued)

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

Figure 6-43: `<stropts.h>` (continued)

```
struct strioctl {
      int      ic_cmd;
      int      ic_timout;
      int      ic_len;
      char     *ic_dp;
};

struct strbuf {
      int      maxlen;
      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];
};
```

Figure 6-43: `<stropts.h>` (continued)

```
#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;
};
```

Figure 6-44: `<termios.h>`

```
#define NCCS                 23
#define CTRL(c)              ((c)&037)

#define IBSHIFT              16
#define _POSIX_VDISABLE      0

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

#define VINTR                0
#define VQUIT                1
#define VERASE               2
#define VKILL                3
#define VEOF                 4
#define VEOL                 5
#define VEOL2                6
#define VMIN                 4
#define VTIME                5
#define VSWTCH               7
#define VSTART               8
#define VSTOP                9
#define VSUSP                10
#define VDSUSP               11
#define VREPRINT             12
#define VDISCARD             13
#define VWERASE              14
#define VLNEXT               15
```

Elements 16-22 of the `C_CC` array are undefined and reserved for future use.

Figure 6-44: `<termios.h>` (continued)

```
#define CNUL            0
#define CDEL            0377
#define CESC            '\\'
#define CINTR           0177
#define CQUIT           034
#define CERASE          '#'
#define CKILL           '@'
#define CEOT            04
#define CEOL            0
#define CEOL2           0
#define CEOF            04
#define CSTART          021
#define CSTOP           023
#define CSWTCH          032
#define CNSWTCH         0
#define CSUSP           CTRL('z')
#define CDSUSP          CTRL('y')
#define CRPRNT          CTRL('r')
#define CFLUSH          CTRL('o')
#define CWERASE         CTRL('w')
#define CLNEXT          CTRL('v')

#define IGNBRK          0000001
#define BRKINT          0000002
#define IGNPAR          0000004
#define PARMRK          0000010
#define INPCK           0000020
#define ISTRIP          0000040
#define INLCR           0000100
#define IGNCR           0000200
#define ICRNL           0000400
#define IUCLC           0001000
#define IXON            0002000
#define IXANY           0004000
#define IXOFF           0010000
```

Figure 6-44: `<termios.h>` (continued)

```
#define OPOST       0000001
#define OLCUC       0000002
#define ONLCR       0000004
#define OCRNL       0000010
#define ONOCR       0000020
#define ONLRET      0000040
#define OFILL       0000100
#define OFDEL       0000200
#define NLDLY       0000400
#define NL0         0
#define NL1         0000400
#define CRDLY       0003000
#define CR0         0
#define CR1         0001000
#define CR2         0002000
#define CR3         0003000
#define TABDLY      0014000
#define TAB0        0
#define TAB1        0004000
#define TAB2        0010000
#define TAB3        0014000
#define BSDLY       0020000
#define BS0         0
#define BS1         0020000
#define VTDLY       0040000
#define VT0         0
#define VT1         0040000
#define FFDLY       0100000
#define FF0         0
#define FF1         0100000
```

Figure 6-44: `<termios.h>` (continued)

```
#define CBAUD           0000017
#define B0              0
#define B50             0000001
#define B75             0000002
#define B110            0000003
#define B134            0000004
#define B150            0000005
#define B200            0000006
#define B300            0000007
#define B600            0000010
#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
```

Figure 6-44: `<termios.h>` (continued)

```
#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          0100000
#define ECHOCTL         0001000
#define ECHOPRT         0002000
#define ECHOKE          0004000
#define FLUSHO          0020000
#define PENDIN          0040000
#define IEXTEN          0000400
#define TIOC            ('T'<<8)

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

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

Figure 6-44: `<termios.h>` (continued)

```
struct termios{
        tcflag_t   c_iflag;
        tcflag_t   c_oflag;
        tcflag_t   c_cflag;
        tcflag_t   c_lflag;
        cc_t       c_cc[NCCS];
};
```

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

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

#define TCL_DEFAULTADDRSZ        4
```

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

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

#define TCO_DEFAULTADDRSZ         4
```

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

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

#define TCOO_DEFAULTADDRSZ        4
```

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

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

typedef long clock_t;
typedef long time_t;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
struct t_info {
         long       addr;
         long       options;
         long       tsdu;
         long       etsdu;
         long       connect;
         long       discon;
         long       servtype;
};
struct t_optmgmt{
         struct netbuf    opt;
         long             flags;
};

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

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

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

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

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

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

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

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

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

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

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

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

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

```
typedef unsigned int greg_t;

#define NGREG 36

typedef greg_t gregset_t[NGREG];

typedef struct fpregset {
         union {
         double        fp_dregs[16];
         float         fp_fregs [32];
         unsigned int  fp_regs[32];
         } fp_r;
         unsigned int fp_csr;
         unsigned int fp_pad;
} fpregset_t;

 typedef struct {
         gregset_t  gregs;
         fpregset_t fpregs;
} mcontext_t;

 typedef struct ucontext{
         unsigned long      uc_flags;
         struct ucontext    *uc_link;
         sigset_t      uc_sigmask;
         stack_t       uc_stack;
         mcontext_t    uc_mcontext;
         long          uc_filler[48];
} ucontext_t;
```

The size of the `ucontext` sruct is 128 words according to the alignment rules in Chapter 3. Specifically, the `fpregset` struct is double word aligned, forcing the `mcontext_t` and `ucontext` structures to also be double word aligned.

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

```
#define CXT_R0          0
#define CXT_AT          1
#define CXT_V0          2
#define CXT_V1          3
#define CXT_A0          4
#define CXT_A1          5
#define CXT_A2          6
#define CXT_A3          7
#define CXT_T0          8
#define CXT_T1          9
#define CXT_T2          10
#define CXT_T3          11
#define CXT_T4          12
#define CXT_T5          13
#define CXT_T6          14
#define CXT_T7          15
#define CXT_S0          16
#define CXT_S1          17
#define CXT_S2          18
#define CXT_S3          19
#define CXT_S4          20
#define CXT_S5          21
#define CXT_S6          22
#define CXT_S7          23
#define CXT_T8          24
#define CXT_T9          25
#define CXT_K0          26
#define CXT_K1          27
#define CXT_GP          28
#define CXT_SP          29
```

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

```
#define CXT_S8          30
#define CXT_RA          31
#define CXT_MDLO        32
#define CXT_MDHI        33
#define CXT_CAUSE       34
#define CXT_EPC         35
```

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

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

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

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

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

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

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

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

#define _POSIX_JOB_CONTROL    1
#define _POSIX_SAVED_IDS      1
#define _POSIX_VDISABLE       *

#define _POSIX_VERSION        *
#define _XOPEN_VERSION        *

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

Figure 6-63: `<unistd.h>` (continued)

```
#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_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_CHOWN_RESTRICTED    7
#define _PC_NO_TRUNC            8
#define _PC_VDISABLE            9

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

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

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

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

```
#define SYS_NMLN  257

struct utsname{
      char        sysname[SYS_NMLN];
      char        nodename[SYS_NMLN];
      char        release[SYS_NMLN];
      char        version[SYS_NMLN];
      char        machine[SYS_NMLN];
      char        m_type[SYS_NMLN];
      char        base_rel[SYS_NMLN];
      char        reserve5[SYS_NMLN];
      char        reserve4[SYS_NMLN];
      char        reserve3[SYS_NMLN];
      char        reserve2[SYS_NMLN];
      char        reserve1[SYS_NMLN];
      char        reserve0[SYS_NMLN];
};
```

The fields `m_type`, `base_rel`, `reserve5`, `reserve4`, `reserve3`, `reserve2`, `reserve1`, and `reserve0` are not defined in the SVID and are reserved for future use.

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

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

#define WSTOPFLG        0177
#define WCONTFLG        0177777
#define WCOREFLG        0200
#define WSIGMASK        0177

#define WWORD(stat)          ((int)((stat))&0177777)

#define WIFEXITED(stat)      ((int) ((stat)&0377)==0)
#define WIFSIGNALED(stat)\
  (((int)((stat)&0377)>0)&&(((int)(((stat)>>8)&0377))==0))
#define WIFSTOPPED(stat)\
  (((int)((stat)&0377)==WSTOPFLAG)&&(((int)(((stat)>>8)\
  &0377))!=0))
#define WIFCONTINUED(stat)  (WWORD(stat)==WCONTFLG)
#define WEXITSTATUS(stat)   (((int)(((stat>>8)&0377))
#define WTERMSIG(stat)      (((int)((stat)&0377)&WSIGMASK))
#define WSTOPSIG(stat)      ((int)(((stat)>>8)&0377))
#define WCOREDUMP(stat)     ((stat)&WCOREFLG)
```

Figure 6-67: `<varargs.h>`

```
typedef char *va_list;
#define va_dcl int va_alist;
#define va_start(list) list = (char *) &va_alist
#define va_end(list)
#define va_arg(list, mode) ((mode *) (list =\
    (char *) ((((int)list + (__builtin_alignof(mode)\
    <=4?3:7)) &(__builtin_alignof(mode)\
    <=4?-4:-8))+sizeof(mode))))[-1]
```

X Window Data Definitions

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

This section contains standard data definitions that describe system data for the optional X Window windowing libraries. 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 6-1: `<X11/Atom.h>`

```
#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 6-1: `<X11/Atom.h>` (continued)

```
#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 6-1: `<X11/Atom.h>` (continued)

```
#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 6-2: <X11/Composite.h>

```
extern WidgetClass compositeWidgetClass;
```

Figure 6-3: <X11/Constraint.h>

```
extern WidgetClass constraintWidgetClass;
```

Figure 6-4: <X11/Core.h>

```
extern WidgetClass coreWidgetClass;
```

Figure 6-5: `<X11/cursorfont.h>`

```
#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 6-5: `<X11/cursorfont.h>` (continued)

```
#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 6-5: `<X11/cursorfont.h>` (continued)

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

Figure 6-6: `<X11/Intrinsic.h>`

```
typedef char                 *String;

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

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

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

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

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

typedef char                *XtPointer;
```

Figure 6-6: `<X11/Intrinsic.h>` (continued)

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

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

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

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

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

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

Figure 6-6: `<X11/Intrinsic.h>` (continued)

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

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

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

typedef XtPointer XtVarArgsList;

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

typedef enum {
      XtCallbackNoList,
      XtCallbackHasNone,
      XtCallbackHasSome
} XtCallbackStatus;

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

Figure 6-6: `<X11/Intrinsic.h>` (continued)

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

typedef enum {
      XtGrabNone,
      XtGrabNonexclusive,
      XtGrabExclusive
} XtGrabKind;

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

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

typedef Boolean         (*XtFilePredicate);
typedef XtPointer       XtRequestId;

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

Figure 6-6: `<X11/Intrinsic.h>` (continued)

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

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

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

Figure 6-6: `<X11/Intrinsic.h>` (continued)

```
#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 6-7: `<X11/Object.h>`

```
extern WidgetClass objectClass;
```

Figure 6-8: `<X11/RectObj.h>`

```
extern WidgetClass rectObjClass;
```

Figure 6-9: `<X11/Shell.h>`

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

Figure 6-10: `<X11/Vendor.h>`

```
extern WidgetClass vendorShellWidgetClass;
```

Figure 6-11: `<X11/X.h>`

```
typedef unsigned long XID;

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

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

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

Figure 6-11: `<X11/X.h>` (continued)

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

Figure 6-11: `<X11/X.h>` (continued)

```
#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
```

Figure 6-11: <X11/X.h> (continued)

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

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

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

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

Figure 6-11: `<X11/X.h>` (continued)

```
#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 6-11: `<X11/X.h>` (continued)

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

#define InputOutput             1
#define InputOnly               2

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

Figure 6-11: `<X11/X.h>` (continued)

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

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

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

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

#define SetModeInsert               0
#define SetModeDelete               1

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

Figure 6-11: `<X11/X.h>` (continued)

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

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

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

Figure 6-11: `<X11/X.h>` (continued)

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

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

#define EvenOddRule                 0
#define WindingRule                 1

#define ClipByChildren              0
#define IncludeInferiors            1

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

#define CoordModeOrigin             0
#define CoordModePrevious           1

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

#define ArcChord                    0
#define ArcPieSlice                 1
```

Figure 6-11: `<X11/X.h>` (continued)

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

#define FontLeftToRight         0
#define FontRightToLeft         1

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

#define AllocNone               0
#define AllocAll                1

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

Figure 6-11: `<X11/X.h>` (continued)

```
#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 6-11: `<X11/X.h>` (continued)

```
#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 6-12: <X11/Xcms.h>

```
#define XcmsFailure              0
#define XcmsSuccess              1
#define XcmsSuccessWithCompression        2

#define XcmsUndefinedFormat
      (XcmsColorFormat)0x00000000
#define XcmsCIEXYZFormat
      (XcmsColorFormat)0x00000001
#define XcmsCIEuvYFormat
      (XcmsColorFormat)0x00000002
#define XcmsCIExyYFormat
      (XcmsColorFormat)0x00000003
#define XcmsCIELabFormat
      (XcmsColorFormat)0x00000004
#define XcmsCIELuvFormat
      (XcmsColorFormat)0x00000005
#define XcmsTekHVCFormat
      (XcmsColorFormat)0x00000006
#define XcmsRGBFormat
      (XcmsColorFormat)0x80000000
#define XcmsRGBiFormat
      (XcmsColorFormat)0x80000001

#define XcmsInitNone             0x00
#define XcmsInitSuccess          0x01

typedef unsigned int XcmsColorFormat;

typedef double XcmsFloat;

typedef struct {
      unsigned short red;
      unsigned short green;
      unsigned short blue;
} XcmsRGB;
```

Figure 6-12: `<X11/Xcms.h>` (continued)

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

typedef struct {
    XcmsFloat X;
    XcmsFloat Y;
    XcmsFloat Z;
} XcmsCIEXYZ;

typedef struct {
    XcmsFloat u_prime;
    XcmsFloat v_prime;
    XcmsFloat Y;
} XcmsCIEuvY;

typedef struct {
    XcmsFloat x;
    XcmsFloat y;
    XcmsFloat Y;
} XcmsCIExyY;

typedef struct {
    XcmsFloat L_star;
    XcmsFloat a_star;
    XcmsFloat b_star;
} XcmsCIELab;
```

Figure 6-12: `<X11/Xcms.h>` (continued)

```
typedef struct {
        XcmsFloat L_star;
        XcmsFloat u_star;
        XcmsFloat v_star;
} XcmsCIELuv;

typedef struct {
        XcmsFloat H;
        XcmsFloat V;
        XcmsFloat C;
} XcmsTekHVC;

typedef struct {
        XcmsFloat pad0;
        XcmsFloat pad1;
        XcmsFloat pad2;
        XcmsFloat pad3;
} XcmsPad;
```

Figure 6-12: `<X11/Xcms.h>` (continued)

```
typedef struct {
     union {
            XcmsRGB              RGB;
            XcmsRGBi             RGBi;
            XcmsCIEXYZ           CIEXYZ;
            XcmsCIEuvY           CIEuvY;
            XcmsCIExyY           CIExyY;
            XcmsCIELab           CIELab;
            XcmsCIELuv           CIELuv;
            XcmsTekHVC           TekHVC;
            XcmsPad              Pad;
            spec;
     unsigned long               pixel;
     XcmsColorFormat             format;
} XcmsColor;

typedef struct {
            XcmsColor            screenWhitePt;
            XPointer             functionSet;
            XPointer             screenData;
            unsigned char        state;
            char                 pad[3];
} XcmsPerScrnInfo;

typedef void *XcmsCCC;

typedef Status (*XcmsConversionProc)();
typedef XcmsConversionProc *XcmsFuncListPtr;
```

Figure 6-12: `<X11/Xcms.h>` (continued)

```
typedef struct {
    char                        *prefix;
    XcmsColorFormat             id;
    XcmsParseStringProc         parseString;
    XcmsFuncListPtr             to_CIEXYZ;
    XcmsFuncListPtr             from_CIEXYZ;
    int                         inverse_flag;
} XcmsColorSpace;

typedef struct {
    XcmsColorSpace              **DDColorSpaces;
    XcmsScreenInitProc          screenInitProc;
    XcmsScreenFreeProc          screenFreeProc;
} XcmsFunctionSet;
```

Figure 6-13: `<X11/Xlib.h>`

```
typedef char *XPointer;

#define Bool                         int
#define Status                       int
#define True                         1
#define False                        0
#define QueuedAlready                0
#define QueuedAfterReading           1
#define QueuedAfterFlush             2

#define AllPlanes         ((unsigned long)~0L)
```

Figure 6-13: `<X11/Xlib.h>` (continued)

```
typdef void XExtData;

typdef void XExtCodes;

typedef struct {
        int depth;
        int bits_per_pixel;
        int scanline_pad;
} XPixmapFormatValues;
```

Figure 6-13: `<X11/Xlib.h>` (continued)

```
typedef struct {
    int function;
    unsigned long plane_mask;
    unsigned long foreground;
    unsigned long background;
    int line_width;
    int line_style;
    int cap_style;
    int join_style;
    int fill_style;
    int fill_rule;
    int arc_mode;
    Pixmap tile;
    Pixmap stipple;
    int ts_x_origin;
    int ts_y_origin;
    Font font;
    int subwindow_mode;
    Bool graphics_exposures;
    int clip_x_origin;
    int clip_y_origin;
    Pixmap clip_mask;
    int dash_offset;
    char dashes;
} XGCValues;

typedef void GC;

typedef void Visual;
```

Figure 6-13: `<X11/Xlib.h>` (continued)

```
typedef void Screen;

typedef struct {
      Pixmap background_pixmap;
      unsigned long background_pixel;
      Pixmap border_pixmap;
      unsigned long border_pixel;
      int bit_gravity;
      int win_gravity;
      int backing_store;
      unsigned long backing_planes;
      unsigned long backing_pixel;
      Bool save_under;
      long event_mask;
      long do_not_propagate_mask;
      Bool override_redirect;
      Colormap colormap;
      Cursor cursor;
} XSetWindowAttributes;
```

Figure 6-13: `<X11/Xlib.h>` (continued)

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

typedef struct {
      int x, y;
      int width, height;
      int border_width;
      int depth;
      Visual *visual;
      Window root;
      int class;
      int bit_gravity;
      int win_gravity;
      int backing_store;
      unsigned long backing_planes;
      unsigned long backing_pixel;
      Bool save_under;
      Colormap colormap;
      Bool map_installed;
      int map_state;
      long all_event_masks;
      long your_event_mask;
      long do_not_propagate_mask;
      Bool override_redirect;
      Screen *screen;
} XWindowAttributes;
```

Figure 6-13: `<X11/Xlib.h>` (continued)

```
typedef struct {
      int family;
      int length;
      char *address;
} XHostAddress;

typedef struct _XImage {
      int width, height;
      int xoffset;
      int format;
      char *data;
      int byte_order;
      int bitmap_unit;
      int bitmap_bit_order;
      int bitmap_pad;
      int depth;
      int bytes_per_line;
      int bits_per_pixel;
      unsigned long red_mask;
      unsigned long green_mask;
      unsigned long blue_mask;
      XPointer obdata;
      struct funcs {
            struct _XImage *(*create_image)();
            int (*destroy_image)();
            unsigned long (*get_pixel)();
            int (*put_pixel)();
            struct _XImage *(*sub_image)();
            int (*add_pixel)();
      } f;
} XImage;
```

Figure 6-13: `<X11/Xlib.h>` (continued)

```
typedef struct {
      int x, y;
      int width, height;
      int border_width;
      Window sibling;
      int stack_mode;
} XWindowChanges;

typedef struct {
      unsigned long pixel;
      unsigned short red, green, blue;
      char flags;
      char pad;
} XColor;

typedef struct {
      short x1, y1, x2, y2;
} XSegment;

typedef struct {
      short x, y;
} XPoint;

typedef struct {
      short x, y;
      unsigned short width, height;
} XRectangle;

typedef struct {
      short x, y;
      unsigned short width, height;
      short angle1, angle2;
} XArc;
```

Figure 6-13: <X11/Xlib.h> (continued)

```
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 void Display;
```

Figure 6-13: `<X11/Xlib.h>` (continued)

```
typedef struct {
      int type;
      unsigned long serial;
      Bool send_event;
      Display *display;
      Window window;
      Window root;
      Window subwindow;
      Time time;
      int x, y;
      int x_root, y_root;
      unsigned int state;
      unsigned int keycode;
      Bool same_screen;
} XKeyEvent;
typedef XKeyEvent XKeyPressedEvent;
typedef XKeyEvent XKeyReleasedEvent;

typedef struct {
      int type;
      unsigned long serial;
      Bool send_event;
      Display *display;
      Window window;
      Window root;
      Window subwindow;
      Time time;
      int x, y;
      int x_root, y_root;
      unsigned int state;
      unsigned int button;
      Bool same_screen;
} XButtonEvent;
typedef XButtonEvent XButtonPressedEvent;
typedef XButtonEvent XButtonReleasedEvent;
```

Figure 6-13: `<X11/Xlib.h>` (continued)

```
typedef struct {
      int type;
      unsigned long serial;
      Bool send_event;
      Display *display;
      Window window;
      Window root;
      Window subwindow;
      Time time;
      int x, y;
      int x_root, y_root;
      unsigned int state;
      char is_hint;
      Bool same_screen;
} XMotionEvent;
typedef XMotionEvent XPointerMovedEvent;

typedef struct {
      int type;
      unsigned long serial;
      Bool send_event;
      Display *display;
      Window window;
      Window root;
      Window subwindow;
      Time time;
      int x, y;
      int x_root, y_root;
      int mode;
      int detail;
      Bool same_screen;
      Bool focus;
      unsigned int state;
} XCrossingEvent;
```

Figure 6-13: `<X11/Xlib.h>` (continued)

```
typedef XCrossingEvent XEnterWindowEvent;
typedef XCrossingEvent XLeaveWindowEvent;
typedef struct {
      int type;
      unsigned long serial;
      Bool send_event;
      Display *display;
      Window window;
      int mode;
      int detail;
} XFocusChangeEvent;
typedef XFocusChangeEvent XFocusInEvent;
typedef XFocusChangeEvent XFocusOutEvent;

typedef struct {
      int type;
      unsigned long serial;
      Bool send_event;
      Display *display;
      Window window;
      char key_vector[32];
} XKeymapEvent;

typedef struct {
      int type;
      unsigned long serial;
      Bool send_event;
      Display *display;
      Window window;
      int x, y;
      int width, height;
      int count;
} XExposeEvent;
```

Figure 6-13: `<X11/Xlib.h>` (continued)

```
typedef struct {
      int type;
      unsigned long serial;
      Bool send_event;
      Display *display;
      Drawable drawable;
      int x, y;
      int width, height;
      int count;
      int major_code;
      int minor_code;
} XGraphicsExposeEvent;

typedef struct {
      int type;
      unsigned long serial;
      Bool send_event;
      Display *display;
      Drawable drawable;
      int major_code;
      int minor_code;
} XNoExposeEvent;

typedef struct {
      int type;
      unsigned long serial;
      Bool send_event;
      Display *display;
      Window window;
      int state;
} XVisibilityEvent;
```

Figure 6-13: `<X11/Xlib.h>` (continued)

```
typedef struct {
      int type;
      unsigned long serial;
      Bool send_event;
      Display *display;
      Window parent;
      Window window;
      int x, y;
      int width, height;
      int border_width;
      Bool override_redirect;
} XCreateWindowEvent;

typedef struct {
      int type;
      unsigned long serial;
      Bool send_event;
      Display *display;
      Window event;
      Window window;
} XDestroyWindowEvent;

typedef struct {
      int type;
      unsigned long serial;
      Bool send_event;
      Display *display;
      Window event;
      Window window;
      Bool from_configure;
} XUnmapEvent;
```

Figure 6-13: `<X11/Xlib.h>` (continued)

```
typedef struct {
      int type;
      unsigned long serial;
      Bool send_event;
      Display *display;
      Window event;
      Window window;
      Bool override_redirect;
} XMapEvent;

typedef struct {
      int type;
      unsigned long serial;
      Bool send_event;
      Display *display;
      Window parent;
      Window window;
} XMapRequestEvent;

typedef struct {
      int type;
      unsigned long serial;
      Bool send_event;
      Display *display;
      Window event;
      Window window;
      Window parent;
      int x, y;
      Bool override_redirect;
} XReparentEvent;
```

Figure 6-13: `<X11/Xlib.h>` (continued)

```
typedef struct {
      int type;
      unsigned long serial;
      Bool send_event;
      Display *display;
      Window event;
      Window window;
      int x, y;
      int width, height;
      int border_width;
      Window above;
      Bool override_redirect;
} XConfigureEvent;

typedef struct {
      int type;
      unsigned long serial;
      Bool send_event;
      Display *display;
      Window event;
      Window window;
      int x, y;
} XGravityEvent;

typedef struct {
      int type;
      unsigned long serial;
      Bool send_event;
      Display *display;
      Window window;
      int width, height;
} XResizeRequestEvent;
```

Figure 6-13: `<X11/Xlib.h>` (continued)

```
typedef struct {
      int type;
      unsigned long serial;
      Bool send_event;
      Display *display;
      Window parent;
      Window window;
      int x, y;
      int width, height;
      int border_width;
      Window above;
      int detail;
      unsigned long value_mask;
} XConfigureRequestEvent;

typedef struct {
      int type;
      unsigned long serial;
      Bool send_event;
      Display *display;
      Window event;
      Window window;
      int place;
} XCirculateEvent;

typedef struct {
      int type;
      unsigned long serial;
      Bool send_event;
      Display *display;
      Window parent;
      Window window;
      int place;
} XCirculateRequestEvent;
```

Figure 6-13: `<X11/Xlib.h>` (continued)

```
typedef struct {
      int type;
      unsigned long serial;
      Bool send_event;
      Display *display;
      Window window;
      Atom atom;
      Time time;
      int state;
} XPropertyEvent;

typedef struct {
      int type;
      unsigned long serial;
      Bool send_event;
      Display *display;
      Window window;
      Atom selection;
      Time time;
} XSelectionClearEvent;

typedef struct {
      int type;
      unsigned long serial;
      Bool send_event;
      Display *display;
      Window owner;
      Window requestor;
      Atom selection;
      Atom target;
      Atom property;
      Time time;
} XSelectionRequestEvent;
```

Figure 6-13: <X11/Xlib.h> (continued)

```
typedef struct {
      int type;
      unsigned long serial;
      Bool send_event;
      Display *display;
      Window requestor;
      Atom selection;
      Atom target;
      Atom property;
      Time time;
} XSelectionEvent;

typedef struct {
      int type;
      Display *display;
      XID resourceid;
      unsigned long serial;
      unsigned char error_code;
      unsigned char request_code;
      unsigned char minor_code;
} XErrorEvent;

typedef struct {
      int type;
      unsigned long serial;
      Bool send_event;
      Display *display;
      Window window;
      Atom message_type;
      int format;
      union {
            char b[20];
            short s[10];
            long l[5];
      } data;
} XClientMessageEvent;
```

Figure 6-13: `<X11/Xlib.h>` (continued)

```
typedef struct {
      int type;
      unsigned long serial;
      Bool send_event;
      Display *display;
      Window window;
      Colormap colormap;
      Bool new;
      int state;
} XColormapEvent;

typedef struct {
      int type;
      unsigned long serial;
      Bool send_event;
      Display *display;
      Window window;
      int request;
      int first_keycode;
      int count;
} XMappingEvent;

typedef struct {
      int type;
      unsigned long serial;
      Bool send_event;
      Display *display;
      Window window;
} XAnyEvent;
```

Figure 6-13: `<X11/Xlib.h>` (continued)

```
typedef union _XEvent {
    int                      type;
    XAnyEvent                xany;
    XKeyEvent                xkey;
    XButtonEvent             xbutton;
    XMotionEvent             xmotion;
    XCrossingEvent           xcrossing;
    XFocusChangeEvent        xfocus;
    XExposeEvent             xexpose;
    XGraphicsExposeEvent     xgraphicsexpose;
    XNoExposeEvent           xnoexpose;
    XVisibilityEvent         xvisibility;
    XCreateWindowEvent       xcreatewindow;
    XDestroyWindowEvent      xdestroywindow;
    XUnmapEvent              xunmap;
    XMapEvent                xmap;
    XMapRequestEvent         xmaprequest;
    XReparentEvent           xreparent;
    XConfigureEvent          xconfigure;
    XGravityEvent            xgravity;
    XResizeRequestEvent      xresizerequest;
    XConfigureRequestEvent   xconfigurerequest;
    XCirculateEvent          xcirculate;
    XCirculateRequestEvent   xcirculaterequest;
    XPropertyEvent           xproperty;
    XSelectionClearEvent     xselectionclear;
    XSelectionRequestEvent   xselectionrequest;
    XSelectionEvent          xselection;
    XColormapEvent           xcolormap;
    XClientMessageEvent      xclient;
    XMappingEvent            xmapping;
    XErrorEvent              xerror;
    XKeymapEvent             xkeymap;
                long         pad[24];
    } XEvent;
```

Figure 6-13: `<X11/Xlib.h>` (continued)

```
#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_byte1;
      unsigned    max_byte1;
      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;
```

Figure 6-13: `<X11/Xlib.h>` (continued)

```
typedef struct {
      char *chars;
      int nchars;
      int delta;
      Font font;
} XTextItem;

typedef struct {
      unsigned char byte1;
      unsigned char byte2;
} XChar2b;

typedef struct {
      XChar2b *chars;
      int nchars;
      int delta;
      Font font;
} XTextItem16;

typedef union {
      Display *display;
      GC gc;
      Visual *visual;
      Screen *screen;
      ScreenFormat *pixmap_format;
      XFontStruct *font;
} XEDataObject;

typedef struct {
      XRectangle       max_ink_extent;
      XRectangle       max_logical_extent;
} XFontSetExtents;

typedef void XFontSet;
```

Figure 6-13: `<X11/Xlib.h>` (continued)

```
typedef struct {
      char              *chars;
      int               nchars;
      int               delta;
      XFontSet    *font_set;
} XmbTextItem;

typedef struct {
      wchar_t     *chars;
      int               nchars;
      int               delta;
      XFontSet    font_set;
} XwcTextItem;

typedef void (*XIMProc)();

typedef void      XIM;
typedef void      XIC;

typedef unsigned long XIMStyle;

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

#define XIMPreeditArea           0x0001L
#define XIMPreeditCallbacks      0x0002L
#define XIMPreeditPosition       0x0004L
#define XIMPreeditNothing        0x0008L
#define XIMPreeditNone           0x0010L
#define XIMStatusArea            0x0100L
#define XIMStatusCallbacks       0x0200L
#define XIMStatusNothing         0x0400L
#define XIMStatusNone            0x0800L
```

Figure 6-13: `<X11/Xlib.h>` (continued)

```
#define XNVaNestedList           "XNVaNestedList"
#define XNQueryInputStyle        "queryInputStyle"
#define XNClientWindow           "clientWindow"
#define XNInputStyle             "inputStyle"
#define XNFocusWindow            "focusWindow"
#define XNResourceName           "resourceName"
#define XNResourceClass          "resourceClass"
#define XNGeometryCallback       "geometryCallback"
#define XNFilterEvents           "filterEvents"
#define XNPreeditStartCallback   "preeditStartCallback"
#define XNPreeditDoneCallback    "preeditDoneCallback"
#define XNPreeditDrawCallback    "preeditDrawCallback"
#define XNPreeditCaretCallback   "preeditCaretCallback"
#define XNPreeditAttributes      "preeditAttributes"
#define XNStatusStartCallback    "statusStartCallback"
#define XNStatusDoneCallback     "statusDoneCallback"
#define XNStatusDrawCallback     "statusDrawCallback"
#define XNStatusAttributes       "statusAttributes"
#define XNArea                   "area"
#define XNAreaNeeded             "areaNeeded"
#define XNSpotLocation           "spotLocation"
#define XNColormap               "colorMap"
#define XNStdColormap            "stdColorMap"
#define XNForeground             "foreground"
#define XNBackground             "background"
#define XNBackgroundPixmap       "backgroundPixmap"
#define XNFontSet                "fontSet"
#define XNLineSpace              "lineSpace"
#define XNCursor                 "cursor"
```

Figure 6-13: `<X11/Xlib.h>` (continued)

```
#define XBufferOverflow         -1
#define XLookupNone             1
#define XLookupChars            2
#define XLookupKeySym           3
#define XLookupBoth             4

typedef XPointer XVaNestedList;

typedef struct {
      XPointer client_data;
      XIMProc callback;
} XIMCallback;

typedef unsigned long XIMFeedback;

#define XIMReverse              1
#define XIMUnderline            (1<<1)
#define XIMHighlight            (1<<2)
#define XIMPrimary              (1<<5)
#define XIMSecondary            (1<<6)
#define XIMTertiary             (1<<7)

typedef struct _XIMText {
      unsigned short length;
      XIMFeedback *feedback;
      Bool encoding_is_wchar;
      union {
            char *multi_byte;
            wchar_t *wide_char;
      } string;
} XIMText
```

Figure 6-13: `<X11/Xlib.h>` (continued)

```
typedef struct _XIMPreeditDrawCallbackStruct {
      int caret;
      int chg_first;
      int chg_length;
      XIMText *text;
} XIMPreeditDrawCallbackStruct;

typedef enum {
      XIMForwardChar, XIMBackwardChar,
      XIMForwardWord, XIMBackwardWord,
      XIMCaretUp, XIMCaretDown,
      XIMNextLine, XIMPreviousLine,
      XIMLineStart, XIMLineEnd,
      XIMAbsolutePosition,
      XIMDontChange
} XIMCaretDirection;

typedef enum {
      XIMIsInvisible,
      XIMIsPrimary,
      XIMIsSecondary
} XIMCaretStyle;

typedef struct _XIMPreeditCaretCallbackStruct {
      int position;
      XIMCaretDirection direction;
      XIMCaretStyle style;
} XIMPreeditCaretCallbackStruct;
```

Figure 6-14: `<X11/Xlib.h>` (continued)

```
typedef enum {
      XIMTextType,
      XIMBitmapType
} XIMStatusDataType;

typedef struct _XIMStatusDrawCallbackStruct {
      XIMStatusDataType type;
      union {
            XIMText *text;
            Pixmap      bitmap;
      } data;
} XIMStatusDrawCallbackStruct;
```

Figure 6-15: <X11/Xresource.h>

```
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 6-15: `<X11/Xresource.h>` (continued)

```
typedef enum {
      XrmoptionNoArg,
      XrmoptionIsArg,
      XrmoptionStickyArg,
      XrmoptionSepArg,
      XrmoptionResArg,
      XrmoptionSkipArg,
      XrmoptionSkipLine,
      XrmoptionSkipNArgs
} XrmOptionKind;

typedef struct {
      char                        *option;
      char                        *specifier;
      XrmOptionKind               argKind;
      XPointer                    value;
} XrmOptionDescRec, *XrmOptionDescList;
```

Figure 6-16: <X11/Xutil.h>

```
#define NoValue         0x0000
#define XValue          0x0001
#define YValue          0x0002
#define WidthValue      0x0004
#define HeightValue     0x0008
#define AllValues       0x000F
#define XNegative       0x0010
#define YNegative       0x0020

typedef struct {
      long flags;
      int x, y;
      int width, height;
      int min_width, min_height;
      int max_width, max_height;
      int width_inc, height_inc;
      struct {
            int x;
            int y;
      } min_aspect, max_aspect;
      int base_width, base_height;
      int win_gravity;
} XSizeHints;

#define USPosition      (1L << 0)
#define USSize          (1L << 1)
#define PPosition       (1L << 2)
#define PSize           (1L << 3)
#define PMinSize        (1L << 4)
#define PMaxSize        (1L << 5)
#define PResizeInc      (1L << 6)
#define PAspect         (1L << 7)
#define PBaseSize       (1L << 8)
#define PWinGravity     (1L << 9)
#define PAllHints (PPosition|PSize|PMinSize| \
      PMaxSize|PResizeInc|PAspect)
```

Figure 6-16: `<X11/Xutil.h>` (continued)

```
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|Icon-
MaskHint|WindowGroupHint)

#define WithdrawnState           0
#define NormalState              1
#define IconicState              3

typedef struct {
      unsigned char              *value;
      Atom                       encoding;
      int                        format;
      unsigned long              nitems;
} XTextProperty;

#define XNoMemory                -1
#define XLocaleNotSupported      -2
#define XConverterNotFound       -3
```

Figure 6-16: `<X11/Xutil.h>` (continued)

```
typedef int XContext;

  typedef enum {
    XStringStyle,
    XCompoundTextStyle,
    XTextStyle,
    XStdICCTextStyle
  } XICCEncodingStyle;

  typedef struct {
    int min_width, min_height;
    int max_width, max_height;
    int width_inc, height_inc;
  } XIconSize;

  typedef struct {
    char *res_name;
    char *res_class;
  } XClassHint;

  #define XDestroyImage(ximage)
    ((*((ximage)->f.destroy_image))((ximage)))
  #define XGetPixel(ximage, x, y)
    ((*((ximage)->f.get_pixel))((ximage), (x), (y)))
  #define XPutPixel(ximage, x, y, pixel)
    ((*((ximage)->f.put_pixel))((ximage), (x),
    (y), (pixel)))
  #define XSubImage(ximage, x, y, width, height)
    ((*((ximage)->f.sub_image))((ximage),
    (x), (y), (width), (height)))
  #define XAddPixel(ximage, value)
    ((*((ximage)->f.add_pixel))((ximage),
    (value)))

  typedef struct _XComposeStatus {
    XPointer compose_ptr;
    int chars_matched;
  } XComposeStatus;
```

Figure 6-16: `<X11/Xutil.h>` (continued)

```
#define IsKeypadKey(keysym)
      (((unsigned)(keysym) >= XK_KP_Space) && \
      ((unsigned)(keysym) <= XK_KP_Equal))
#define IsCursorKey(keysym)
      (((unsigned)(keysym) >= XK_Home) && \
      ((unsigned)(keysym) < XK_Select))
#define IsPFKey(keysym)
      (((unsigned)(keysym) >= XK_KP_F1) \
      && ((unsigned)(keysym) <= XK_KP_F4))
#define IsFunctionKey(keysym)
      (((unsigned)(keysym) >= XK_F1) && \
((unsigned)(keysym) <= XK_F35))
#define IsMiscFunctionKey(keysym)
      (((unsigned)(keysym) >= XK_Select) && \
      ((unsigned)(keysym) <= XK_Break))
#define IsModifierKey(keysym)
      ((((unsigned)(keysym) >= XK_Shift_L) \
      && ((unsigned)(keysym) <= XK_Hyper_R))
      || ((unsigned)(keysym) == XK_Mode_switch)
      || ((unsigned)(keysym) == XK_Num_Lock))

typedef void Region;

#define RectangleOut    0
#define RectangleIn     1
#define RectanglePart   2

typedef struct {
      Visual *visual;
      VisualID visualid;
      int   screen;
      int   depth;
      int   class;
      unsigned long red_mask;
      unsigned long green_mask;
      unsigned long blue_mask;
      int   colormap_size;
      int   bits_per_rgb;
} XVisualInfo;
```

Figure 6-16: `<X11/Xutil.h>` (continued)

```
#define VisualNoMask              0x0
#define VisualIDMask              0x1
#define VisualScreenMask          0x2
#define VisualDepthMask           0x4
#define VisualClassMask           0x8
#define VisualRedMaskMask         0x10
#define VisualGreenMaskMask       0x20
#define VisualBlueMaskMask        0x40
#define VisualColormapSizeMask    0x80
#define VisualBitsPerRGBMask      0x100
#define VisualAllMask             0x1FF

typedef struct {
      Colormap                    colormap;
      unsigned long               red_max;
      unsigned long               red_mult;
      unsigned long               green_max;
      unsigned long               green_mult;
      unsigned long               blue_max;
      unsigned long               blue_mult;
      unsigned long               base_pixel;
      VisualID                    visualid;
      XID                         killid;
} XStandardColormap;

#define ReleaseByFreeingColormap ((XID) 1L)
#define BitmapSuccess             0
#define BitmapOpenFailed          1
#define BitmapFileInvalid         2
#define BitmapNoMemory            3
#define XCSUCCESS                 0
#define XCNOMEM                   1
#define XCNOENT                   2
```

TCP/IP Data Definitions

This section is new, but will not be diffmarked.

Figure 6-17: `<netinet/in.h>`

NOTE 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.

```
#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 6-18: `<netinet/ip.h>`

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

Figure 6-19: `<X11/netinet/tcp.h>`

```
#define       TCP_NODELAY   0x01
```

Development Environment

Development Commands

NOTE THE FACILITIES AND INTERFACES DESCRIBED IN THIS SECTION ARE OPTIONAL COMPONENTS OF THE *System V Application Binary Interface*.

NOTE *This chapter is new, but will not be marked with diff-marks.*

The Development Environment for MIPS implementations of System V Release 4 will contain all of the development commands required by the System V ABI, namely;

`as`	`cc`	`ld`
`m4`	`lex`	`yacc`

Each command accepts all of the options required by the System V ABI, as defined in the SD_CMD section of the *System V Interface Definition, Third Edition*.

PATH Access to Development Tools

The development environment for the MIPS 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 MIPS implementations of the System V ABI shall include each of the following commands as defined in the AS_CMD section of the *System V Interface Definition, Third Edition*.

`pkgproto` `pkgtrans` `pkgmk`

System Headers

Systems that do not have an ABI Development Environment may or 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 depend only 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 MIPS development environments.

Execution Environment

Application Environment

NOTE *This chapter is new, but will not be marked with diff-marks.*

This section specifies the execution environment information available to application programs running on a MIPS ABI-conforming computer.

The /dev Subtree

All networking device files described in the Generic ABI shall be supported on all MIPS ABI-conforming computers. In addition, the following device files are required to be present on all MIPS 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/sxtXX`
`/dev/ttyXX` These device files, where XX represents a two-digit integer, represent device entries for terminal sessions. All these device files must be examined by the `ttyname()` call. Applications must not have the device names of individual terminals hard-coded within them. The `sxt` entries are optional in the system but, if present must be included in the library routine's search.