SYSTEM V APPLICATION BINARY INTERFACE

Motorola 68000 Processor Family Supplement

UNIX Software Operation

Published by Prentice-Hall, Inc.
A Division of Simon & Schuster
Englewood Cliffs, New Jersey 07632

IMPORTANT NOTE TO USERS

While every effort has been made to ensure the accuracy of all information in this document, AT&T assumes no liability to any party for any loss or damage caused by errors or omissions or by statements of any kind in this document, its updates, supplements, or special editions, whether such errors are omissions or statements resulting from negligence, accident, or any other cause. AT&T further assumes no liability arising out of the application or use of any product or system described herein; nor any liability for incidental or consequential damages arising from the use of this document. AT&T disclaims all warranties regarding the information contained herein, whether expressed, implied or statutory, *including implied warranties of merchantability or fitness for a particular purpose.* AT&T makes no representation that the interconnection of products in the manner described herein will not infringe on existing or future patent rights, nor do the descriptions contained herein imply the granting or license to make, use or sell equipment constructed in accordance with this description.

AT&T reserves the right to make changes without further notice to any products herein to improve reliability, function, or design.

TRADEMARKS

UNIX is a registered trademark of AT&T.

The publisher offers discounts on this book when ordered in bulk quantities.
For more information, write:

Special Sales
Prentice-Hall, Inc.
College Technical and Reference Division
Englewood Cliffs, New Jersey 07632

or

call 201-592-2498

For single copies, call 201-767-5937

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-877663-6

UNIX PRESS

A Prentice Hall Title

Contents

Figures and Tables

1. INTRODUCTION

1. INTRODUCTION

1 INTRODUCTION

Motorola 68000 Family and the System V ABI

The **System V Application Binary Interface**, or **ABI**, defines a system interface for compiled application programs. Its purpose is to establish a standard binary interface for application programs on systems that implement UNIX System V Release 4.0 or some other operating system that complies with the **System V Interface Definition, Issue 3**.

This document is a supplement to the generic **System V ABI**, and it contains information specific to System V implementations built on the Motorola 68000 processor architecture family. The generic term "Motorola 68000 family" is used in this specification to mean the Motorola MC68020, MC68030, and MC68040 processor architectures; it does not refer to the MC68000, MC68008, or MC68010.

Together, these two specifications, the generic **System V ABI** and the **Motorola 68000 Family System V ABI Supplement**, constitute a complete *System V Application Binary Interface* specification for systems that implement the architecture of the Motorola 68000 family.

How to Use the Motorola 68000 Family ABI Supplement

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

As with the **System V ABI**, this specification references other publicly-available reference documents, including the

- **MC68020 32-Bit Microprocessor User's Manual**, MC68020UM/AD
- **MC68030 Enhanced 32-Bit Microprocessor User's Manual**, MC68030UM/AD
- **MC68040 32-Bit Microprocessor User's Manual**, MC68040UM/AD
- **M68000 Programmer's Reference Manual**, M68000PM/AD

available from Motorola.

All the information referenced by this supplement should be considered part of this specification, and just as binding as the requirements and data explicitly included here.

Evolution of the ABI Specification

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

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

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

2. SOFTWARE INSTALLATION

2. SOFTWARE INSTALLATION

2 SOFTWARE INSTALLATION

Software Distribution Formats

Physical Distribution Media

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

- 5.25-inch floppy disk: 96 TPI (80 tracks/side) doubled-sided, 15 sectors/track, 512 bytes/sector, total format capacity of 1.2 megabytes per disk.
- 3.5-inch floppy disk: 135 TPI (80 tracks/side) double sided, 18 sectors/track, 512 bytes/sector, total format capacity of 1.44 megabytes per disk.
- 1/2-inch reel-to-reel tape: conforms to ANSI-standard reel-to-reel tape standard which consists of 9 tracks, 1600 BPI, no label.
- 60 MB quarter-inch cartridge tape in QIC-24 format.

The QIC-24 cartridge tape data format is described in *Proposed Standard for Data Interchange on the Streaming 1/4 Inch Magnetic Tape Cartridge Using Group Code Recording at 10000 FRPI*, 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.

3. LOW LEVEL SYSTEM INFORMATION

3 LOW-LEVEL SYSTEM INFORMATION

Machine Interface

Processor Architecture

The *MC68020 32-Bit Microprocessor User's Manual*, the *MC68030 Enhanced 32-Bit Microprocessor User's Manual*, and the *MC68040 32-Bit Microprocessor User's Manual* define the 68000 family processor architecture. The *M68000 Programmer's Reference Manual* defines the programming model. An MC68851 Paged Memory Management Unit (PMMU) may be present with the MC68020. An MC68881 or MC68882 Floating Point Coprocessor (FPCP) is assumed to be present with the MC68020 or MC68030. Programs intended to execute directly on the processor use the instruction set, instruction encodings, and instruction semantics of this architecture. A program may use only the instructions defined for the MC68040. Refer to the *M68000 Programmer's Reference Manual* for information regarding proper instruction usage for the MC68020, MC68030, and MC68040 processors.

To be ABI-conforming, the processor must implement the architecture's 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. A software emulation of the architecture could conform to the ABI.

Some processors might support the 68000 and MC68881/2 FPCP architectures as subsets, providing additional instructions or capabilities. Programs that use those capabilities explicitly do not conform to the 68000 ABI. Executing those programs on machines without the additional capabilities gives undefined behavior.

Data Representation

Fundamental Types

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

Figure 3-1: Scalar Types

Type	C	sizeof	Alignment (bytes)	68000
Integral	`signed char` `char`	1	1	signed byte
	`unsigned char`	1	1	unsigned byte
	`short` `signed short`	2	2	signed word (2 bytes)
	`unsigned short`	2	2	unsigned word

	`int`
	`signed int`
	`long`
	`signed long`
	`enum`
		4	4	signed long word (4 bytes)

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

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

NOTE If a `double` or `long double` appears on the stack, not as a part of an aggregate or union, then it is aligned to 4 bytes.

Aggregates and Unions

An array assumes the alignment of its elements' type. The size of any object, including arrays, structures, and unions, always is a multiple of the object's alignment. Structure and union objects may, therefore, require padding to meet size and alignment constraints.

- The alignment of a structure or a union is the maximum of the alignment of its elements.
- Each member is assigned to the lowest available offset with the appropriate alignment. This may require *internal padding*, depending on the previous member.
- A structure's size is increased, if necessary, to make it a multiple of the structure's alignment. This may require *tail padding*, depending on the last member.

Aggregates or unions residing on the stack only require 4-byte alignment.

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

Figure 3-2: Structure Smaller Than a Long Word

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

Byte aligned, `sizeof` is 1

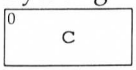


Figure 3-3: No Padding

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

Long word aligned, `sizeof` is 8

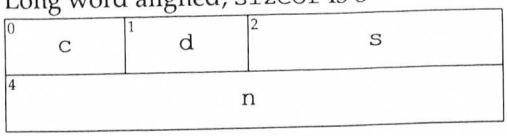


Figure 3-4: Internal Padding

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

Word aligned, `sizeof` is 4

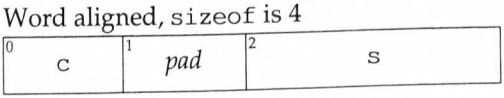


Figure 3-5: Internal and Tail Padding

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

8-byte aligned (4-byte on stack), `sizeof` is 24

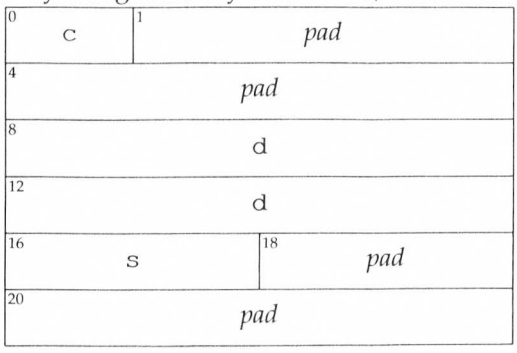
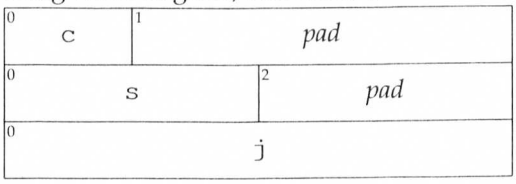


Figure 3-6: `union` Allocation

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

Long word aligned, `sizeof` is 4



Bit-Fields

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

Figure 3-7: Bit-Field Ranges

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

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

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

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

Figure 3-8: Bit Numbering

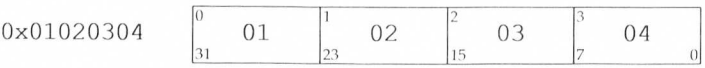


Figure 3-9: Left-to-Right Allocation

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

Long word aligned, `sizeof` is 4

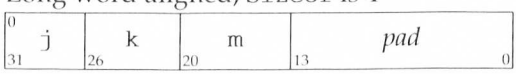


Figure 3-10: Boundary Alignment

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

Long word aligned, `sizeof` is 12

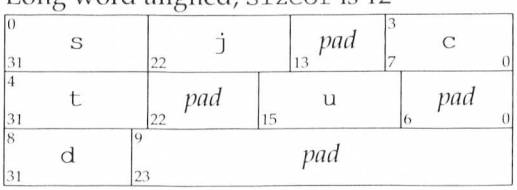


Figure 3-11: Storage Unit Sharing

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

Word aligned, `sizeof` is 2

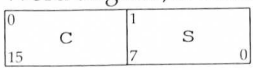


Figure 3-12: `union` Allocation

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

Word aligned, `sizeof` is 2

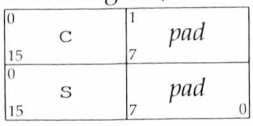
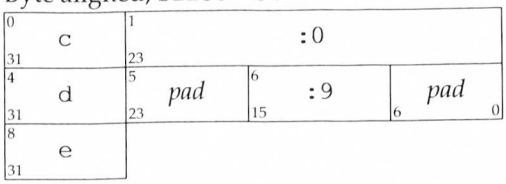


Figure 3-13: Unnamed Bit-Fields

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

Byte aligned, `sizeof` is 9



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

Function Calling Sequence

This section discusses the standard function calling sequence, including stack frame layout, register usage, parameter passing, etc. The operating system interface and C programs use this calling sequence. See "Coding Examples" later in this chapter for more information on C.

NOTE

All of the coding examples in this section are simply illustrations of a sample implementation.

Registers and the Stack Frame

The 68000 provides 8 data and 8 address registers, which are global to a running program. Additionally, the MC68040, or the MC68881/2 Floating Point Coprocessor, provides 8 global floating-point registers. Brief register descriptions appear in Figure 3-14; more complete information appears later.

Figure 3-14: Processor Registers

Type	Names		Usage
68000	%d0 %d1 %a0 %a1		Scratch registers
	%d2 ... %d7 %a2 ... %a5		Local register variables
	%a6	%fp	Frame pointer (if implemented)
	%a7	%sp	Stack pointer
		%pc	Program counter
		%ccr	Condition code register
MC68040, MC68881/2	%fp0 %fp1		Scratch registers
	%fp2 ... %fp7		Local register variables
		%fpcr	Floating Point Control Register
		%fpsr	Floating Point Status Register
		%fpiar	Floating Point Instruction Address Register

In addition to the registers, each function has a frame on the run-time stack. This stack grows downward from high addresses. Figure 3-15 shows the stack frame organization right after function prologue processing has allocated the frame and saved the registers (see below).

Figure 3-15: Standard Stack Frame

Base	Offset	Contents	Purpose	
	+4+4*n	argument long word *n*	incoming arguments	*High Addresses*
		...		
	+4	argument long word 0		
		return address		
%fp	(optional)	previous %fp (optional)		
		unspecified	local storage and register save area	
		...		
SP	(SP after prologue)	variable size		
		stack top, unused		*Low addresses*

Several key points about the stack frame deserve mention.

- The stack frame is long word aligned.
- Functions may save registers %d2 through %d7, %a2 through %a6, and %fp2 through %fp7 *as necessary*, without saving unused registers.
- All incoming arguments reside on the stack. "Coding Examples" below explains how variable argument lists may be implemented.
- A frame pointer may be implemented.
- Other areas depend on the compiler and the code being compiled. The standard calling sequence does not define a maximum stack frame size, nor does it restrict how a language system uses the "unspecified" areas of the standard stack frame.

Registers %d2 through %d7, %a2 through %a6, and %fp2 through %fp7, which are visible to both a calling and a called function, "belong" to the calling function. In other words, a called function must save these registers' values before it changes them, restoring their values before it returns. Remaining registers "belong" to the called function. If a calling function wants to preserve such a register value

across a function call, it must save the value in its local stack frame.

Across function boundaries, the standard function prologue saves registers %d2 through %d7, %a2 through %a6, and %fp2 through %fp7 (as needed) and allocates stack space, including the required areas of Figure 3-15 and any private space it needs. The example below illustrates this, saving registers %fp, %d7, %a5, and %fp2 and allocating 80 bytes for local storage.

Figure 3-16: Function Prologue

```
fcn:
        link.l      %fp,&-80
        movm.l      %d7/%a5,-(%sp)
        fmovm.x     %fp2,-(%sp)
```

The movm instruction manipulates registers as part of the normal function prologue and epilogue. As explained later, the function epilogue executes a movm instruction to unwind the stack and restore the saved registers to their original condition.

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

Some registers have assigned roles.

%fp or %a6 — The *frame pointer*, if used by an individual function, holds the address of the local storage within a stack frame, referenced as negative offsets from %fp.

%sp or %a7 — The *stack pointer* holds the limit of the current stack frame, which is the address of the stack's topmost valid long word. The long word to which %sp points is "in" the valid stack.

%d0	*Integral return values* appear in %d0.
%a0	*Pointer return values* appear in %a0. When calling a function that returns a structure or union, the caller allocates space for the return value and sets %a0 to its address. A function that returns a structure or union value places the same address in %a0 before it returns.
%fp0	*Floating-point return values* appear in this register.

Except as specified here, %d0, %d1, %a0, %a1, %fp0, and %fp1 are scratch registers. Functions do not need to preserve their values for the caller.

Local registers %d2 through %d7, %a2 through %a5, and %fp2 through %fp7 have no specified role in the standard calling sequence.

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

Integral and Pointer Arguments

As mentioned, a function receives all integral and pointer argument long words on the stack. Functions pass all integer-valued arguments as long words, expanding signed or unsigned bytes and words as needed.

NOTE

The following examples use the frame pointer. Functions not using %fp would find arguments at different locations.

Figure 3-17: Integral and Pointer Arguments

Call	Argument	Callee
`g(1, 2, 3, (void *)0);`	1	8(%fp)
	2	12(%fp)
	3	16(%fp)
	(void *)0	20(%fp)

Floating-Point Arguments

The stack also holds floating-point arguments: single-precision values use one long word, double-precision use two, and extended-precision use four. The example below uses only double-precision arguments.

Figure 3-18: Floating-Point Arguments

Call	Argument	Callee
`h(1.414, 1, 2.998e10);`	long word 0, 1.414	8(%fp)
	long word 1, 1.414	12(%fp)
	1	16(%fp)
	long word 0, 2.998e10	20(%fp)
	long word 1, 2.998e10	24(%fp)

Structure and Union Arguments

As described in the data representation section, structures and unions always have the alignment of their most strictly aligned member. When passed as arguments, sizes are rounded up to the next long word size. Structure and union objects appear directly on the stack, occupying as many long words as necessary.

Figure 3-19: Structure and Union Arguments

Call	Argument	Callee
`i(1, s);`	1	8(%fp)
	long word 0, s	12(%fp)
	long word 1, s	16(%fp)
	...	...

Functions Returning Scalars or No Value

A function that returns an integral value places its result in `%d0`. A function that returns a pointer value places its result in `%a0`. A function that returns a floating-point value places its result in `%fp0`.

Functions that return no value put no specified value in any return register.

Just as the function prologue may save registers, the epilogue must restore those same registers before returning to the caller. To complete the example from Figure 3-16, the following function epilogue restores `%d7`, `%a5`, and `%fp2` and then returns.

Figure 3-20: Function Epilogue

```
fmovm.x     (%sp)+,%fp2
movm.l      (%sp)+,%d7/%a5
unlk        %fp
rts
```

Functions Returning Structures or Unions

As mentioned above, when a function returns a structure or union, it expects the caller to provide space for the return value and to place its address in register %a0. Having the caller supply the return object's space allows re-entrancy.

NOTE

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

A function returning a structure or union also sets %a0 to the value it finds in %a0. Thus when the caller receives control again, the address of the returned object resides in register %a0. Both the calling and the called functions must cooperate to pass the return value successfully. Failure of either side to meet its obligations leads to undefined program behavior.

The following example assumes the return object has been copied, and its address is in register %a5.

Figure 3-21: Function Epilogue

```
mov.l      %a5,%a0
fmovm.x    (%sp)+,%fp2
movm.l     (%sp)+,%d7/%a5
unlk       %fp
rts
```

Operating System Interface

Virtual Address Space

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

Page Size

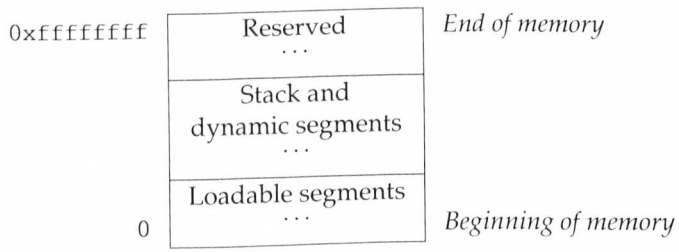
Memory is organized by pages, which are the system's smallest units of memory allocation. Page size can vary from one system to another, depending on the processor, memory management unit and system configuration. Allowable page sizes are 2K, 4K, or 8K. Processes may call `sysconf(BA_OS)` to determine the system's current page size.

Virtual Address Assignments

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

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

Figure 3-22: Virtual Address Configuration



CAUTION

Programs that dereference null pointers are erroneous. Although such programs may appear to work on the 68000, they might fail or behave differently on other systems. To enhance portability, programmers are strongly cautioned not to rely on dereferencing null pointers.

Loadable segments

Processes' loadable segments may begin at 0. The exact addresses depend on the executable file format (see Chapters 4 and 5).

Stack and dynamic segments

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

Reserved

A reserved area resides at the top of virtual space.

NOTE

Although application programs may begin at virtual address 0, they conventionally begin above 0x10000 (64K), leaving the initial 64K with an invalid address mapping. Processes that reference this invalid memory (for example, by dereferencing a null pointer) generate an access exception trap, as described in the "Exception Interface" section of this chapter.

As the figure shows, the system reserves the high end of virtual space, with a process's stack and dynamic segments below that. Although the exact boundary between the reserved area and a process depends on the system's configuration, the reserved area shall not consume more than 512 MB from the virtual address

space. Thus the user virtual address range has a minimum upper bound of `0xdfffffff`. Individual systems may reserve less space, increasing processes' virtual memory range. More information follows in the section "Managing the Process Stack."

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

Managing the Process Stack

Section "Process Initialization" in this chapter describes the initial stack contents. Stack addresses can change from one system to the next—even from one process execution to the next on a single system. Processes, therefore, should *not* depend on finding their stack at a particular virtual address. The stack segment has read and write permissions.

A tunable configuration parameter controls the system maximum stack size. A process also can use `setrlimit`(BA_OS), to set its own maximum stack size, up to the system limit. Changes in the stack virtual address and size affect the virtual addresses for dynamic segments. Consequently, processes should *not* depend on finding their dynamic segments at particular virtual addresses. Facilities exist to let the system choose dynamic segment virtual addresses.

Coding Guidelines

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

Processes' address spaces typically have three segment areas that can change size from one execution to the next: the stack [through `setrlimit`(BA_OS)], the data segment [through `malloc`(BA_OS)], and the dynamic segment area [through `mmap`(KE_OS)]. Consequently, an address that is available in one process

execution might not be available in the next. A program that used `mmap`(KE_OS) to request a mapping at a specific address thus could appear to work in some environments and fail in others. For this reason, programs that wish to establish a mapping in their address space should let the system choose the address.

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

Processor Execution Modes

Two execution modes exist in the 68000 architecture: user and supervisor. Processes run in user mode (the least privileged). The operating system kernel runs in supervisor mode. A program executes the `trap` instruction to change execution modes.

NOTE

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

Exception Interface

As the 68000 user manuals describe, the processor also changes mode to handle *exceptions*. Exceptions can be explicitly generated by a process as a result of instruction execution. The operating system defines the following correspondence between hardware exceptions and the signals specified by `signal`(BA_OS).

Figure 3-23: Exceptions and Signals

Exception Name	Signal
trap #1 (breakpoint trap)	SIGTRAP
external memory fault	see below
address error	SIGBUS
all floating-point exceptions	SIGFPE
illegal instruction	SIGILL
integer zero-divide	SIGFPE
privileged opcode	SIGILL
trace	SIGTRAP
chk, chk2 instruction	SIGFPE
cptrapcc, trapcc, trapv	SIGFPE
line 1010 emulator	SIGSYS
line 1111 emulator	SIGSYS
trap #2-15	SIGSYS

An external memory fault exception can generate various signals, depending on why the exception occurred.

SIGBUS or SIGEMT
: The process accessed a memory location in a way disallowed by the current mapping's permissions. As an example, the process tried to store a value into a location without write permission.

SIGSEGV
: The process referenced a memory address for which no valid mapping existed.

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. As an example, a C program begins executing at a function named `main`, conventionally declared in the following way.

Figure 3-24: Declaration for `main`

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

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

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

Registers

Registers `%d0` through `%d7`, `%a0` through `%a6`, and all MC68040 or MC68881/2 FPCP floating-point data registers have unspecified values at process entry. The floating-point control registers are set to provide IEEE default behavior. Consequently, a program that requires registers to have specific values must set them explicitly during process initialization. It should *not* rely on the operating system to set all registers to 0. See "Process Stack" below for information about the initial values of the stack registers.

As the architecture defines, the status register controls and monitors the processor. Application programs cannot access the entire status register directly; they run in the processor's *user mode*, and the instructions to access the entire status register are privileged. Nonetheless, a program can access the condition code register, which initially has the following value.

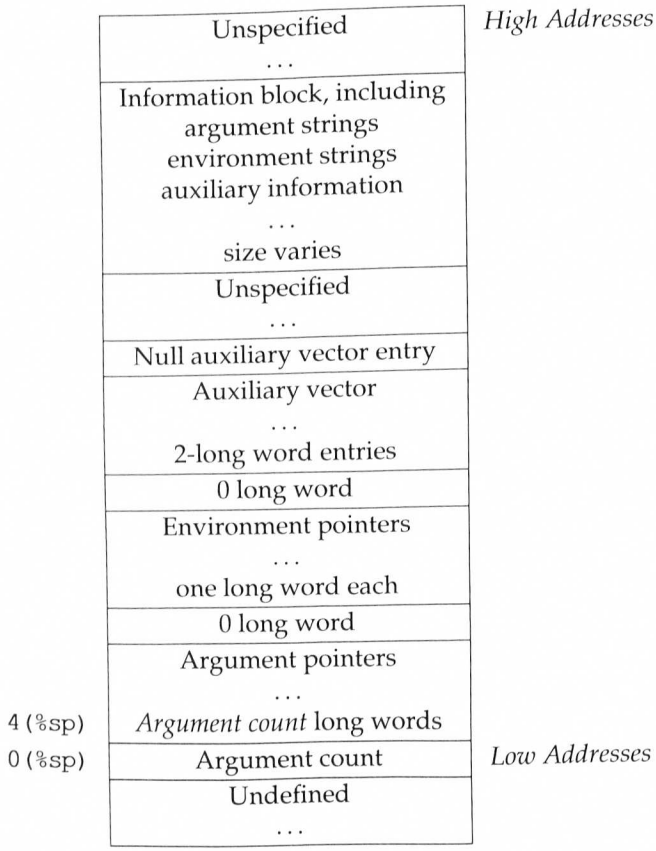
Figure 3-25: **Condition Code Register Fields**

Field	Value	Note
XNZVC	unspecified	Condition codes unspecified

Process Stack

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

Figure 3-26: Initial Process Stack



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

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

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

Figure 3-27: Auxiliary Vector

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

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

Name	Value	a_un
AT_NULL	0	ignored
AT_IGNORE	1	ignored
AT_EXECFD	2	a_val
AT_PHDR	3	a_ptr
AT_PHENT	4	a_val
AT_PHNUM	5	a_val
AT_PAGESZ	6	a_val

Figure 3-28: Auxiliary Vector Types, a_type (continued)

Name	Value	a_un
AT_BASE	7	a_ptr
AT_FLAGS	8	a_val
AT_ENTRY	9	a_ptr

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

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

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

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

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

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

AT_PAGESZ	If present, this entry's a_val member gives the system page size, in bytes. The same information also is available through sysconf(BA_OS).
AT_BASE	The a_ptr member of this entry holds the base address at which the interpreter program was loaded into memory. See "Program Header" in the System V ABI for more information about the base address.
AT_FLAGS	If present, the a_val member of this entry holds one-bit flags. Bits with undefined semantics are set to zero. No flags are defined for the 68000.
AT_ENTRY	The a_ptr member of this entry holds the entry point of the application program to which the interpreter program should transfer control.

Other auxiliary vector types are reserved.

In the following example, the stack resides at an address below 0xf0000000, growing toward lower addresses. The process receives three arguments.

- cp
- src
- dst

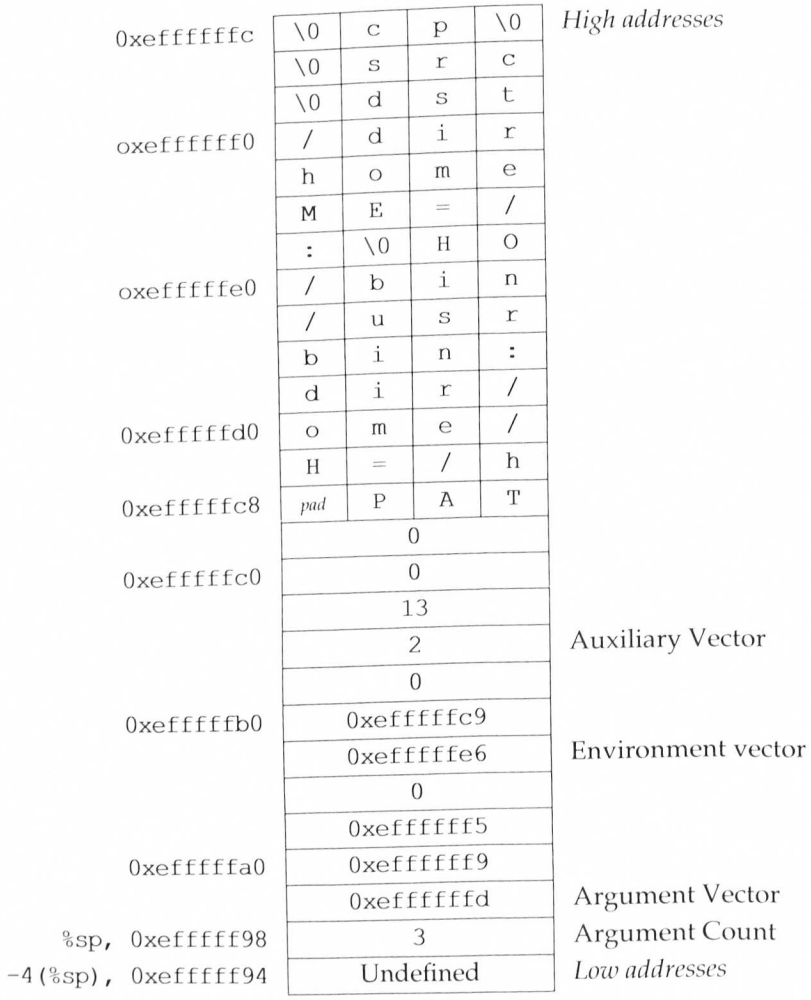
It also inherits two environment strings (this example is not intended to show a fully configured execution environment).

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

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

- {AT_EXECFD, 13}

Figure 3-29: Example Process Stack



Coding Examples

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

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

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

The following sections describe the differences between these models. Code sequences for the models (when different) appear together, allowing easier comparison.

NOTE

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

NOTE

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

Code Model Overview

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

Position-independent code relies on two techniques.

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

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

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

Assembly language examples below show the explicit notation needed for position-independent code.

name@GOT — This expression denotes the displacement in the global offset table of the entry for the symbol *name*.

name@PLT — This expression denotes the displacement in the procedure linkage table of the entry for the symbol *name*.

name@GOTPC

This expression denotes a PC-relative reference to the global offset table entry for the symbol *name*.

name@PLTPC

This expression denotes a PC-relative reference to the procedure linkage table entry for the symbol *name*.

Position-Independent Function Prologue

This section describes the function prologue for position-independent code. A function's prologue first allocates the local stack space. Position-independent functions also set register %a5 to the global offset table's address, accessed with the symbol _GLOBAL_OFFSET_TABLE_. Because %a5 is private for each function and preserved across function calls, a function calculates its value once at the entry.

NOTE As a reminder, this entire section contains examples. Using %a5 is a convention, not a requirement; moreover, this convention is private to a function. Not only could other registers serve the same purpose, but different functions in a program could use different registers.

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

Figure 3-30: Position-Independent Function Prologue

```
name:
   link.l   %fp,&-80
   movm.l   %a5,-(%sp)
   lea      (%pc,_GLOBAL_OFFSET_TABLE_@GOTPC),%a5
```

Data Objects

This discussion excludes stack-resident objects, because programs always compute their virtual addresses relative to the stack and frame pointers. Instead, this section describes objects with static storage duration. Symbolic references in absolute code put the symbols' values—or absolute virtual addresses—into instructions.

Figure 3-31: Absolute Load and Store

C

```
extern int src;
extern int dst;
extern int *ptr;
ptr = &dst;

*ptr = src;
```

Assembly

```
.global   src,dst,ptr


mov.l     &dst,ptr

mov.l     src,([ptr])
```

Position-independent code cannot contain absolute addresses. Referencing global symbols must be done with a base register and global offset table index (as in these examples). Alternatively, instructions that reference symbols hold the PC-relative offsets into the global offset table. Combining the offset with the PC gives the absolute address of the table entry holding the desired address.

Figure 3-32: Position-Independent Load and Store

C

```
extern int src;
extern int dst;
extern int *ptr;
ptr = &dst;

*ptr = src;
```

Assembly

```
.global src,dst,ptr


mov.l   (%a5,dst@GOT),([%a5,ptr@GOT])

mov.l   ([%a5,ptr@GOT]),%a4
mov.l   ([%a5,src@GOT]),(%a4)
```

Function Calls

Function calls in absolute code put the symbols' values—or absolute virtual addresses—into instructions. Programs use the `jsr` or `bsr` instructions to make function calls. For absolute code, the destination operand is an absolute address. Even when the code for a function resides in a shared object, the caller uses the same assembly language instruction sequence, although in that case control passes from the original call, through an indirection sequence, to the desired destination. See "Procedure Linkage Table" in Chapter 5 for more information on the indirection sequence.

Figure 3-33: Absolute Direct Function Call

C

```
extern void function();
function();
```

Assembly

```
.global   function
jsr       function
```

For position-independent code, the `bsr` instruction's destination operand is a PC-relative value. Typically, the destination will be an entry in the procedure linkage table mentioned above.

Figure 3-34: Position-Independent Direct Function Call

C

```
extern void function();
function();
```

Assembly

```
.global   function
bsr       function@PLTPC
```

Indirect function calls also use the `jsr` instruction.

Figure 3-35: Absolute Indirect Function Call

C

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

(*ptr)();
```

Assembly

```
.global   ptr,name

mov.l     &name,ptr

jsr       ([ptr])
```

For position-independent code, the global offset table supplies absolute addresses for all required symbols, whether the symbols name objects or functions.

Figure 3-36: Position-Independent Indirect Function Call

C

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

(*ptr)();
```

Assembly

```
.global   ptr,name

mov.l     (%a5,name@GOT),([%a5,ptr@GOT])

mov.l     ([%a5,ptr@GOT]),%a0
jsr       (%a0)
```

Branching

Programs use branch instructions to control their execution flow. As defined by the architecture, branch instructions can hold a PC-relative value with a range that covers the entire address space.

Figure 3-37: Branch Instruction, Both Models

C

```
label:
          ...
          goto label;
```

Assembly

```
.L01:
          ...
          bra.l    .L01
```

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

- The selection expression resides in register `%d0`;
- `case` label constants begin at zero;
- `case` labels, `default`, and the address table use assembly names `.Lcase`*i*, `.Ldef`, and `.Ltab`, respectively.

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

Figure 3-38: Absolute `switch` Code

C

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

Assembly

```
        cmp.l   %d0,&3
        bhi     .Ldef
        asl.l   &2,%d0
        jmp     ([%pc,%d0.l,.Ltab])
        ...
.Ltab:  .long   .Lcase0
        .long   .Ldef
        .long   .Lcase2
        .long   .Lcase3
```

Figure 3-39: Position-Independent `switch` Code

C

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

Assembly

```
            cmp.l   %d0,&3
            bhi     .Ldef
            mov.l   (%pc,%d0.w*4,.Ltab),%d0
            jmp     (%pc,%d0,.Ltab)
            ...
.Ltab:      .long   .Lcase0-.Ltab
            .long   .Ldef-.Ltab
            .long   .Lcase2-.Ltab
            .long   .Lcase3-.Ltab
```

C Stack Frame

Figure 3-40 shows the C stack frame organization. It conforms to the standard stack frame with designated roles for unspecified areas in the standard frame.

Figure 3-40: C Stack Frame

Base	Offset	Contents	Purpose	
	+4+4*n	argument long word *n*	incoming arguments	*High Addresses*
		...		
	+4	argument long word 0		
SP'	(SP before call)	return address		
		unspecified	local storage and register save area	
		...		
SP	(SP after call)	variable size		
		outgoing arguments		
		stack top, unused		*Low addresses*

Variable Argument List

Previous sections describe the rules for passing arguments. Unfortunately, some otherwise portable C programs depend on the argument passing scheme, implicitly assuming that 1) all arguments reside on the stack, and 2) arguments appear in increasing order on the stack. Programs that make these assumptions never have been portable, but they have worked on many machines, including the 68000. Nonetheless, portable C programs should use the header files `<stdarg.h>` or `<varargs.h>` to deal with variable argument lists (on 68000 and other machines as well).

Allocating Stack Space Dynamically

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

Figure 3-40 shows the layout of the C stack frame. The double line divides the area allocated by the compiler from the dynamically allocated memory. Dynamic space is allocated below the line as a downward growing heap whose size changes as required. Typical C functions have no space in the heap. All areas above the double line in the current frame have a known size to the compiler. Dynamic stack allocation thus takes the following steps.

1. Stack frames are long word aligned; dynamic allocation should preserve this property. Thus the program rounds (up) the desired byte count to a multiple of 4.
2. The program decreases the stack pointer by the rounded byte count, increasing its frame size. At this point, the "new" space resides just below the register save area at the bottom of the stack.

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

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

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

Existing stack objects reside at fixed offsets from the frame pointer; stack heap allocation doesn't 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.

4. OBJECT FILES

4. OBJECT FILES

4 OBJECT FILES

ELF Header

Machine Information

For file identification in e_ident, the Motorola 68000 Family requires the following values.

Figure 4-1: Motorola 68000 Family Identification, e_ident

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

The ELF header's e_flags member holds bit flags associated with the file. Motorola 68000 Family defines no flags; so this member contains zero. Processor identification resides in the ELF header's e_machine member and must have the value 4, defined as the name EM_68K.

Sections

Special Sections

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

Figure 4-2: Special Sections

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

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

.plt — This section holds the procedure linkage table. See "Procedure Linkage Table" in Chapter 5 for more information.

Symbol Table

Symbol Values

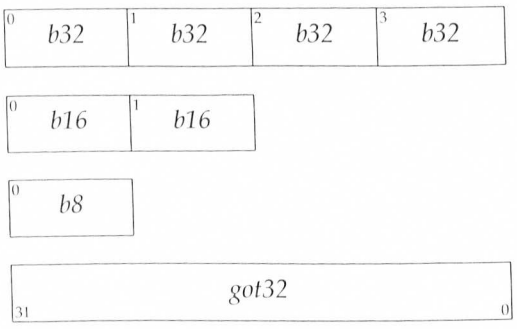
If an executable file contains a reference to a function defined in one of its associated shared objects, the symbol table section for that file will contain an entry for that symbol. The `st_shndx` member of that symbol table entry contains `SHN_UNDEF`. This signals to the dynamic linker that the symbol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

Relocation

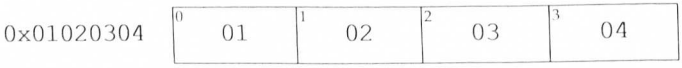
Relocation Types

Relocation entries describe how to alter the following instruction and data fields (bit numbers appear in the lower box corners; byte numbers appear in the upper box corners).

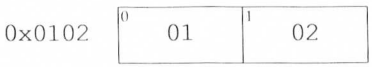
Figure 4-3: Relocatable Fields



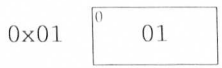
b32 This specifies a 32-bit field occupying 4 bytes with arbitrary alignment. These values use the byte order illustrated below.



b16 This specifies a 16-bit field occupying 2 bytes with arbitrary alignment.



b8 This specifies a 8-bit field occupying 1 byte with arbitrary alignment.



got32 This specifies a 32-bit field occupying 4 bytes with long word alignment. These bytes represent values in the same byte order as *b32*.

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

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

B This means the base address at which a shared object has been loaded into memory during execution. Generally, a shared object file is built with a 0 base virtual address, but the execution address will be different.

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

G′ This means the address of entry zero in the global offset table.

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

L′ This means the address of entry zero in the procedure linkage table.

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

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

A relocation entry's `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 the Motorola 68000 Family uses only `Elf32_Rela` relocation entries, the relocation table entry holds the addend. In all cases, the addend and the computed result use the same byte order.

Figure 4-4: Relocation Types

Name	Value	Field	Calculation
R_68K_NONE	0	none	*none*
R_68K_32	1	*b32*	S + A
R_68K_16	2	*b16*	S + A
R_68K_8	3	*b8*	S + A
R_68K_PC32	4	*b32*	S + A - P
R_68K_PC16	5	*b16*	S + A - P
R_68K_PC8	6	*b8*	S + A - P
R_68K_GOT32	7	*b32*	G + A - P
R_68K_GOT16	8	*b16*	G + A - P
R_68K_GOT8	9	*b8*	G + A - P
R_68K_GOT32O	10	*b32*	G - G′
R_68K_GOT16O	11	*b16*	G - G′
R_68K_GOT8O	12	*b8*	G - G′
R_68K_PLT32	13	*b32*	L + A - P
R_68K_PLT16	14	*b16*	L + A - P
R_68K_PLT8	15	*b8*	L + A - P
R_68K_PLT32O	16	*b32*	L - L′
R_68K_PLT16O	17	*b16*	L - L′
R_68K_PLT8O	18	*b8*	L - L′
R_68K_COPY	19	none	*none*
R_68K_GLOB_DAT	20	*got32*	S
R_68K_JMP_SLOT	21	*got32*	S
R_68K_RELATIVE	22	*b32*	B + A

Some relocation types have semantics beyond simple calculation.

R_68K_GOT32 — This relocation type resembles R_68K_PC32, except it refers to the address of the symbol's global offset table entry and additionally instructs the link editor to build a global offset table.

R_68K_GOT16 — This relocation type resembles R_68K_PC16, except it refers to the address of the symbol's global offset table entry and additionally instructs the link editor to build a global offset table.

R_68K_GOT8 — This relocation type resembles R_68K_PC8, except it refers to the address of the symbol's global offset table entry and additionally instructs the link editor to build a global offset table.

R_68K_GOT32O — This relocation type resembles R_68K_GOT32, except it refers to the address of the symbol's global offset table entry relative to the address of entry zero in the GOT.

R_68K_GOT16O — This relocation type resembles R_68K_GOT16, except it refers to the address of the symbol's global offset table entry relative to the address of entry zero in the GOT.

R_68K_GOT8O — This relocation type resembles R_68K_GOT8, except it refers to the address of the symbol's global offset table entry relative to the address of entry zero in the GOT.

R_68K_PLT32 — This relocation type resembles R_68K_PC32, except it refers to the address of the symbol's procedure linkage table entry and additionally instructs the link editor to build a procedure linkage table.

R_68K_PLT16 — This relocation type resembles R_68K_PC16, except it refers to the address of the symbol's procedure linkage table entry and additionally instructs the link editor to build a procedure linkage table.

R_68K_PLT8 — This relocation type resembles R_68K_PC8, except it refers to the address of the symbol's procedure linkage table entry and additionally instructs the link editor to build a procedure linkage table.

R_68K_PLT32O — This relocation type resembles R_68K_PLT32, except it refers to the address of the symbol's procedure linkage table entry relative to the address of entry zero in the PLT.

R_68K_PLT16O — This relocation type resembles R_68K_PLT16, except it refers to the address of the symbol's procedure linkage table entry relative to the address of entry zero in the PLT.

R_68K_PLT8O — This relocation type resembles R_68K_PLT8, except it refers to the address of the symbol's procedure linkage table entry relative to the address of entry zero in the PLT.

R_68K_COPY — This relocation type assists dynamic linking. Its offset member refers to a location in a writable segment. The symbol table index specifies a symbol that should exist both in the current object file and in a shared object. During execution, the dynamic linker copies data associated with the shared object's symbol to the location specified by the offset.

R_68K_GLOB_DAT — This relocation type resembles R_68K_32, except it is used to set a global offset table entry to the specified symbol's value. The relocation type allows one to determine the correspondence between symbols and global offset table entries. The relocated field should be aligned on a long word boundary. This relocation type does *not* use the addend.

R_68K_JMP_SLOT — This relocation type assists dynamic linking. Its offset member gives the location of a global offset table entry. This relocation type does *not* use the addend.

R_68K_RELATIVE — This relocation type assists dynamic linking. The addend member contains a value representing a relative address within a shared object. The offset member gives a location within the shared object for the final virtual address. The dynamic linker computes the corresponding virtual address by adding the virtual address at which the shared object was loaded to the relative address. Relocation entries for this type must specify 0 for the symbol table index.

5. PROGRAM LOADING AND DYNAMIC LINKING

5. PROGRAM LOADING AND DYNAMIC LINKING

5 PROGRAM LOADING AND DYNAMIC LINKING

Program Loading

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

Virtual addresses and file offsets for Motorola 68000 Family segments are congruent modulo 8 K (0x2000) or larger powers of 2. Because 8 KB is the maximum page size, the files will be suitable for paging regardless of physical page size. Figure 5-1 is an example of an executable file.

Figure 5-1: Executable File

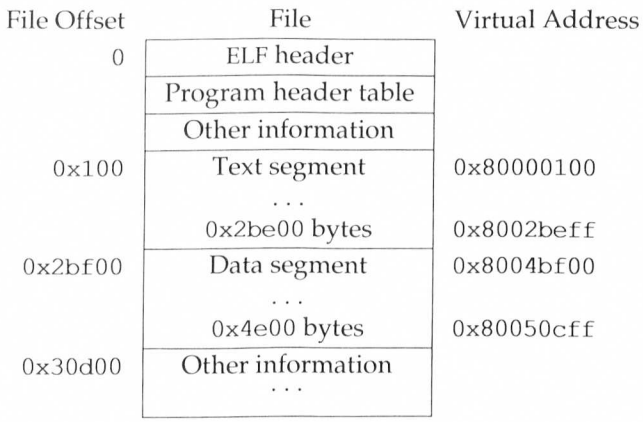


Figure 5-2: Program Header Segments

Member	Text	Data
p_type	PT_LOAD	PT_LOAD
p_offset	0x100	0x2bf00
p_vaddr	0x80000100	0x8004bf00
p_paddr	unspecified	unspecified
p_filesz	0x2be00	0x4e00
p_memsz	0x2be00	0x5e24
p_flags	PF_R+PF_X	PF_R+PF_W+PF_X
p_align	0x2000	0x2000

Although the example's file offsets and virtual addresses are congruent modulo 8 K for both text and data, up to four file pages hold impure text or data (depending on page size and file system block size).

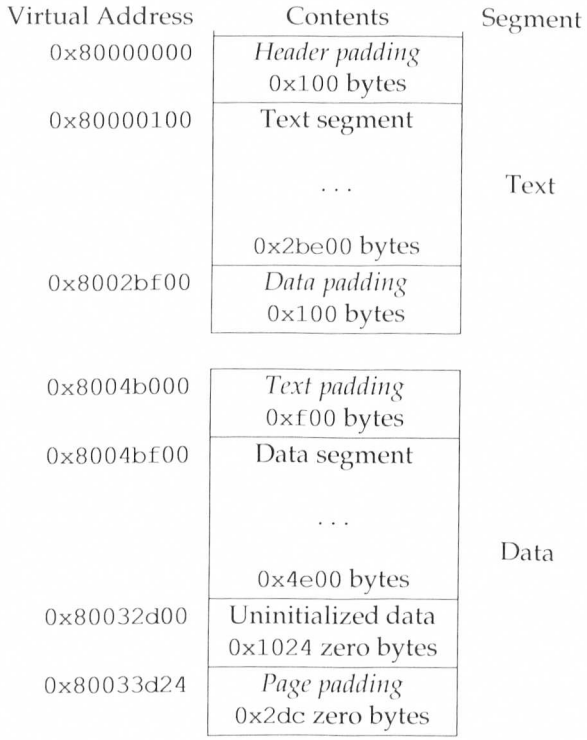
- The first text page contains the ELF header, the program header table, and other information.
- The last text page holds a copy of the beginning of data.
- The first data page has a copy of the end of text.
- The last data page may contain file information not relevant to the running process.

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

The end of the data segment requires special handling for uninitialized data, which the system defines to begin with zero values. Thus if a file's last data page includes information not in the logical memory page, the extraneous data must be set to zero, not the unknown contents of the executable file. "Impurities" in

the other three pages are not logically part of the process image; whether the system expunges them is unspecified. The memory image for this program follows, assuming 4 KB (0x1000) pages.

Figure 5-3: Process Image Segments



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

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

Figure 5-4: Example Shared Object Segment Addresses

Source	Text	Data	Base Address
File	0x200	0x2a400	0x0
Process 1	0xc0080200	0xc00aa400	0xc0080000
Process 2	0xc0082200	0xc00ac400	0xc0082000
Process 3	0xd00c0200	0xd00ea400	0xd00c0000
Process 4	0xd00c6200	0xd00f0400	0xd00c6000

Dynamic Linking

Dynamic Section

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

DT_PLTGOT — On the 68000, this entry's d_ptr member gives the address of the first entry in the global offset table. As mentioned below, the first three global offset table entries are reserved, and two are used to hold procedure linkage table information.

Global Offset Table

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

Initially, the global offset table holds information as required by its relocation entries [see "Relocation" in Chapter 4]. When the dynamic linker creates memory segments for a loadable object file, it processes the relocation entries, some of which will be type R_68K_GLOB_DAT referring to the global offset table. The dynamic linker determines the associated symbol values, calculates their absolute addresses, and sets the appropriate memory table entries to the proper values. Although the absolute addresses are unknown when the link editor builds an object file, the dynamic linker knows the addresses of all memory segments and can thus calculate the absolute addresses of the symbols contained therein.

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

are available during execution.

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

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

A global offset table's format and interpretation are processor-specific. For the Motorola 68000 Family an offset into the table is an *unsigned* value, allowing only non-negative "subscripts" into the array of addresses.

Function Addresses

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

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

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

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

Procedure Linkage Table

Much as the global offset table redirects position-independent address calculations to absolute locations, the procedure linkage table redirects position-independent function calls to absolute locations. The link editor cannot resolve execution transfers (such as function calls) from one executable or shared object to another. Consequently, the link editor arranges to have the program transfer control to entries in the procedure linkage table. On the 68000, procedure linkage tables reside in shared text, but they use addresses in the private global offset table. The dynamic linker determines the destinations' absolute addresses and modifies the global offset table's memory image accordingly. The dynamic linker thus can redirect the entries without compromising the position-independence and sharability of the program's text. Executable files and shared object files have separate procedure linkage tables.

Figure 5-5: Initial Procedure Linkage Table

```
.PLT0:   mov.l   got_plus_4,-(%sp)
         jmp     ([got_plus_8])
         nop
         nop
.PLT1:   jmp     ([name1@GOTPC,%pc])
         mov.l   &offset,-(%sp)
         bra     .PLT0
.PLT2:   jmp     ([name2@GOTPC,%pc])
         mov.l   &offset,-(%sp)
         bra     .PLT0
```

Following the steps below, the dynamic linker and the program "cooperate" to resolve symbolic references through the procedure linkage table and the global offset table.

1. When first creating the memory image of the program, the dynamic linker sets the second and the third entries in the global offset table to special values. Steps below explain more about these values.
2. For illustration, assume the program calls `name1`, which transfers control to the label `.PLT1`.
3. The first instruction jumps to the address in the global offset table entry for `name1`. Initially, the global offset table holds the address of the following instruction, not the real address of `name1`.
4. Consequently, the program pushes a relocation offset (*offset*) on the stack. The relocation offset is a 32-bit, non-negative byte offset into the relocation table. The designated relocation entry will have type `R_68K_JMP_SLOT`, and its offset will specify the global offset table entry used in the previous `jmp` instruction. The relocation entry also contains a symbol table index, thus telling the dynamic linker what symbol is being referenced, `name1` in this case.

5. After pushing the relocation offset, the program then jumps to `.PLT0`, the first entry in the procedure linkage table. The `mov.l` instruction places the value of the second global offset table entry (*got_plus_4*) on the stack, thus giving the dynamic linker one long word of identifying information. The program then jumps to the address in the third global offset table entry (*got_plus_8*), which transfers control to the dynamic linker.

6. When the dynamic linker receives control, it unwinds the stack, looks at the designated relocation entry, finds the symbol's value, stores the "real" address for `name1` in its global offset table entry, and transfers control to the desired destination.

7. Subsequent executions of the procedure linkage table entry will transfer directly to `name1`, without calling the dynamic linker a second time. That is, the `jmp` instruction at `.PLT1` will transfer to `name1`, instead of "falling through" to the next instruction.

The `LD_BIND_NOW` environment variable can change dynamic linking behavior. If its value is non-null, the dynamic linker evaluates procedure linkage table entries before transferring control to the program. That is, the dynamic linker processes relocation entries of type `R_68K_JMP_SLOT` during process initialization. Otherwise, the dynamic linker evaluates procedure linkage table entries lazily, delaying symbol resolution and relocation until the first execution of a table entry.

NOTE

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

6. LIBRARIES

6. LIBRARIES

6 LIBRARIES

System Library

Additional Entry Points

There are no additional entry points required by the Motorola 68000 Family Processor Supplement.

Support Routines

Besides operating system services, **libsys** contains the following processor-specific support routine. The routine is also accessible named with a leading underscore.

Figure 6-1: libsys Support Routines

sbrk

```
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 location 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 space 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.

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 Motorola 68000 Family processor architecture. Declarations for the data objects listed below can be found in the Data Definitions section of this chapter or immediately following the table.

Figure 6-2: libsys, Global External Data Symbols

`__flt_rounds`	`__huge_val`

Application Constraints

As described above, **libsys** provides symbols for applications. In a few cases, however, an executable is obliged to provide symbols for the library. In addition to the application-provided symbols listed in this section of the **System V ABI**, conforming applications on the Motorola 68000 Family processor architecture are also required to provide the following symbols.

`extern _end;`

This symbol refers neither to a routine nor to a location with interesting contents. Instead, its address must correspond to the beginning of a program's dynamic allocation area, called the heap. Typically, the heap begins immediately after the data segment of the program's executable file. This value is normally provided by the static linker.

`extern const int _lib_version;`

This variable's value specifies the compilation and execution mode for the program. If the value is zero, the program wants to preserve the semantics of older (pre-ANSI) C, where conflicts exist with ANSI. Otherwise, the value is non-zero, and the program wants ANSI C semantics. This value is normally provided by the compiler.

C Library

Additional Support Routines

There are no additional support routines required by the Motorola 68000 Family Processor Supplement.

System Data Interfaces

Data Definitions

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

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

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

Figure 6-3: <assert.h>

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

Figure 6-4: <ctype.h>

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

extern unsigned char    __ctype[];

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

Figure 6-5: <dirent.h>

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

Figure 6-6: <errno.h>, **Part 1 of 4**

```
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
#define EROFS          30
#define EMLINK         31
#define EPIPE          32
```

Figure 6-7: <errno.h>, Part 2 of 4

```
#define EDOM            33
#define ERANGE          34
#define ENOMSG          35
#define EIDRM           36
#define ECHRNG          37
#define EL2NSYNC        38
#define EL3HLT          39
#define EL3RST          40
#define ELNRNG          41
#define EUNATCH         42
#define ENOCSI          43
#define EL2HLT          44
#define EDEADLK         45
#define ENOLCK          46
#define ENOSTR          60
#define ENODATA         61
#define ETIME           62
#define ENOSR           63
#define ENONET          64
#define ENOPKG          65
#define EREMOTE         66
#define ENOLINK         67
#define EADV            68
#define ESRMNT          69
#define ECOMM           70
#define EPROTO          71
```

Figure 6-8: <errno.h>, **Part 3 of 4**

```
#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       158
#define ESTALE          162

/* The following errno values are optional. */

#define EWOULDBLOCK     EDEADLK
#define EBADE           50
#define EBADR           51
#define EXFULL          52
#define ENOANO          53
#define EBADRQC         54
#define EBADSLT         55
#define EDEADLOCK       56
#define EBFONT          57
#define EDOTDOT         76
#define ELIBACC         83
#define ELIBBAD         84
#define ELIBSCN         85
```

Figure 6-9: `<errno.h>`, **Part 4 of 4**

```
#define ELIBMAX         86
#define ELIBEXEC        87
#define EINPROGRESS     128
#define EALREADY        129
#define ENOTSOCK        130
#define EDESTADDRREQ    131
#define EMSGSIZE        132
#define EPROTOTYPE      133
#define ENOPROTOOPT     134
#define EPROTONOSUPPORT         135
#define ESOCKTNOSUPPORT         136
#define EOPNOTSUPP      137
#define EPFNOSUPPORT    138
#define EAFNOSUPPORT    139
#define EADDRINUSE      140
#define EADDRNOTAVAIL   141
#define ENETDOWN        142
#define ENETUNREACH     143
#define ENETRESET       144
#define ECONNABORTED    145
#define ECONNRESET      146
#define ENOBUFS         147
#define EISCONN         148
#define ENOTCONN        149
#define ESHUTDOWN       150
#define ETOOMANYREFS    151
#define ETIMEDOUT       152
#define ECONNREFUSED    153
#define EHOSTDOWN       156
#define EHOSTUNREACH    157
#define EPROCLIM        159
#define EUSERS          160
#define EDQUOT          161
#define EPOWERFAIL      163
```

Figure 6-10: `<fcntl.h>`, **Part 1 of 2**

```
#define O_RDONLY        0
#define O_WRONLY        1
#define O_RDWR          2
#define O_NDELAY        04
#define O_APPEND        010
#define O_SYNC          020
#define O_NONBLOCK      0100
#define O_CREAT         00400
#define O_TRUNC         01000
#define O_EXCL          02000
#define O_NOCTTY        04000

#define F_DUPFD         0
#define F_GETFD         1
#define F_SETFD         2
#define F_GETFL         3
#define F_SETFL         4
#define F_GETLK         14
#define F_SETLK         6
#define F_SETLKW        7
#define F_FREESP        11

#define FD_CLOEXEC      1
#define O_ACCMODE       03
```

Figure 6-11: <fcntl.h>, **Part 2 of 2**

```
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 6-12: <float.h>

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

Figure 6-13: <fmtmsg.h>

```
#define MM_NULL         0L

#define MM_HARD         0x00000001L
#define MM_SOFT         0x00000002L
#define MM_FIRM         0x00000004L
#define MM_RECOVER      0x00000100L
#define MM_NRECOV       0x00000200L
#define MM_APPL         0x00000008L
#define MM_UTIL         0x00000010L
#define MM_OPSYS        0x00000020L
#define MM_PRINT        0x00000040L
#define MM_CONSOLE      0x00000080L
#define MM_NOSEV        0
#define MM_HALT         1
#define MM_ERROR        2
#define MM_WARNING      3
#define MM_INFO         4

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

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

Figure 6-14: <ftw.h>

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

#define FTW_F           0
#define FTW_D           1
#define FTW_DNR         2
#define FTW_NS          3
#define FTW_SL          4
#define FTW_DP          6
#define FTW_SLN         7

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

Figure 6-15: <grp.h>

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

Figure 6-16: <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_ALLOC       0100000

#define IPC_PRIVATE     (key_t)0

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

Figure 6-17: `<langinfo.h>`**, Part 1 of 2**

```
#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 6-18: `<langinfo.h>`, **Part 2 of 2**

```
#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 6-19: <limits.h>

```
#define MB_LEN_MAX      5

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

/* the #undef-fed values vary and should be
        retrieved using sysconf() or pathconf() */

#define _POSIX_ARG_MAX          4096
#define _POSIX_CHILD_MAX        6
#define _POSIX_LINK_MAX         8
#define _POSIX_MAX_CANON        255
#define _POSIX_MAX_INPUT        255
#define _POSIX_NAME_MAX         14
#define _POSIX_NGROUPS_MAX      0
#define _POSIX_OPEN_MAX         16
#define _POSIX_PATH_MAX         255
#define _POSIX_PIPE_BUF         512

#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 FCHAR_MAX       1048576
```

Figure 6-20: <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;
} lconv;

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

Figure 6-21: <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 6-22: <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 6-23: <mon.h>

```
struct hdr {
        char    *lpc;
        char    *hpc;
        int     nfns;
};

struct cnt {
        char    *fnpc;
        long    mcnt;
};

typedef unsigned short WORD;
```

Figure 6-24: <sys/mount.h>

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

Figure 6-25: <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_susec;
        time_t         msg_rtime;
        long           msg_rusec;
        time_t         msg_ctime;
        long           msg_cusec;
        long           pad[4];
};

#define MSG_NOERROR    010000
```

Figure 6-26: <netconfig.h>, **Part 1 of 2**

```
struct  netconfig {
        char            *nc_netid;
        unsigned long    nc_semantics;
        unsigned long    nc_flag;
        char            *nc_protofmly;
        char            *nc_proto;
        char            *nc_device;
        unsigned long    nc_nlookups;
        char           **nc_lookups;
        unsigned long    nc_unused[8];
};
#define NC_TPI_CLTS             1
#define NC_TPI_COTS             2
#define NC_TPI_COTS_ORD         3
#define NC_TPI_RAW              4
#define NC_NOFLAG               00
#define NC_VISIBLE              01
#define NC_BROADCAST            02
```

Figure 6-27: `<netconfig.h>`**, Part 2 of 2**

```
#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 6-28: <netdir.h>

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

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

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

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

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

Figure 6-29: <nl_types.h>

```
#define NL_SETD         1

typedef short nl_item ;

typedef void *nl_catd;
```

Figure 6-30: <sys/param.h>

```
#define HZ              sysconf(3)

#define NGROUPS_UMIN    0

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

#define NADDR           13

#define NBBY            8
#define NBPSCTR         512
```

Figure 6-31: <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 6-32: <sys/procset.h>

```
#define P_INITPID       1
#define P_INITUID       0
#define P_INITPGID      0
#define P_MYID          (-1)

typedef long id_t;

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

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

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

Figure 6-33: <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 6-34: <sys/regset.h>

```
typedef int     greg_t;
#define NGREG   18
typedef greg_t  gregset_t[NGREG];

#define R_D0    0
#define R_D1    1
#define R_D2    2
#define R_D3    3
#define R_D4    4
#define R_D5    5
#define R_D6    6
#define R_D7    7
#define R_A0    8
#define R_A1    9
#define R_A2    10
#define R_A3    11
#define R_A4    12
#define R_A5    13
#define R_A6    14
#define R_A7    15
#define R_SP    15
#define R_PC    16
#define R_PS    17

typedef struct fpregset {
        int     f_pcr;
        int     f_psr;
        int     f_fpiaddr;
        int     f_fpregs[8][3];
} fpregset_t;
```

Figure 6-35: <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

struct rlimit {
        rlim_t  rlim_cur;
        rlim_t  rlim_max;
};

typedef unsigned long rlim_t;
```

Figure 6-36: <rpc.h>, **Part 1 of 12**

```
#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];
};
```

Figure 6-37: `<rpc.h>`**, Part 2 of 12**

```
struct opaque_auth {
  int    oa_flavor;
  char   *oa_base;
  unsigned int oa_length;
};

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

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;
};
```

Figure 6-38: <rpc.h>, **Part 3 of 12**

```
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

#define DES_FAILED(err)         ((err) > DESERR_NOHWDEVICE)
```

Figure 6-39: <rpc.h>, Part 4 of 12

```
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_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_UDERROR=23,
        RPC_TLIERROR=20,
        RPC_FAILED=16
};
#define RPC_PMAPFAILURE RPC_RPCBFAILURE
```

Figure 6-40: <rpc.h>, **Part 5 of 12**

```
#define _RPC_NONE               0
#define _RPC_NETPATH            1
#define _RPC_VISIBLE            2
#define _RPC_CIRCUIT_V          3
#define _RPC_DATAGRAM_V         4
#define _RPC_CIRCUIT_N          5
#define _RPC_DATAGRAM_N         6
#define _RPC_TCP                7
#define _RPC_UDP                8

#define RPC_ANYSOCK     -1
#define RPC_ANYFD       RPC_ANYSOCK

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

Figure 6-41: <rpc.h>, **Part 6 of 12**

```
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 6-42: <rpc.h>, **Part 7 of 12**

```
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 6-43: <rpc.h>, **Part 8 of 12**

```
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;
};

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

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 6-44: <rpc.h>, Part 9 of 12

```
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 6-45: <rpc.h>, **Part 10 of 12**

```
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 6-46: <rpc.h>, Part 11 of 12

```
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;
  u_long window;
};

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

Figure 6-47: `<rpc.h>`, **Part 12 of 12**

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

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

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

Figure 6-48: <search.h>

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

Figure 6-49: <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;
        char            sem_pad[2];
        unsigned short  sem_nsems;
        time_t          sem_otime;
        long            sem_ousec;
        time_t          sem_ctime;
        long            sem_cusec;
        long            pad[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 6-50: <setjmp.h>

```
#define _JBLEN 13
#define _SIGJBLEN 64
typedef int jmp_buf[_JBLEN];
typedef int sigjmp_buf[_SIGJBLEN];
```

Figure 6-51: <sys/shm.h>

```
struct shmid_ds {
        struct ipc_perm       shm_perm;
        int             shm_segsz;
        struct anon_map       *shm_amp;
        unsigned short shm_lkcnt;
        char            pad[2];
        pid_t           shm_lpid;
        pid_t           shm_cpid;
        unsigned long   shm_nattch;
        unsigned long   shm_cnattch;
        time_t          shm_atime;
        long            shm_ausec;
        time_t          shm_dtime;
        long            shm_dusec;
        time_t          shm_ctime;
        long            shm_cusec;
        long            pad1[4];
};
#define SHMLBA          sysconf(31)

#define SHM_RDONLY      010000
#define SHM_RND         020000
```

Figure 6-50: <setjmp.h>

```
#define _JBLEN 13
#define _SIGJBLEN 64
typedef int jmp_buf[_JBLEN];
typedef int sigjmp_buf[_SIGJBLEN];
```

Figure 6-51: <sys/shm.h>

```
struct shmid_ds {
        struct ipc_perm       shm_perm;
        int             shm_segsz;
        struct anon_map       *shm_amp;
        unsigned short shm_lkcnt;
        char            pad[2];
        pid_t           shm_lpid;
        pid_t           shm_cpid;
        unsigned long   shm_nattch;
        unsigned long   shm_cnattch;
        time_t          shm_atime;
        long            shm_ausec;
        time_t          shm_dtime;
        long            shm_dusec;
        time_t          shm_ctime;
        long            shm_cusec;
        long            pad1[4];
};
#define SHMLBA          sysconf(31)

#define SHM_RDONLY      010000
#define SHM_RND         020000
```

Figure 6-52: <sigaction.h>

```
struct sigaction {
        void            (*sa_handler)();
        sigset_t        sa_mask;
        int             sa_flags;
};

#define SA_NOCLDSTOP    0x00000001
#define SA_NOCLDWAIT    0x00000002
#define SA_ONSTACK      0x00010000
#define SA_RESETHAND    0x00020000
#define SA_RESTART      0x00040000
#define SA_SIGINFO      0x00080000
#define SA_NODEFER      0x00100000
```

Figure 6-53: <sys/siginfo.h>, **Part 1 of 3**

```
#define SI_FROMUSER(sip)  ((sip)->si_code <= 0)
#define SI_FROMKERNEL(sip)  ((sip)->si_code > 0)
#define SI_USER         0

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

#define FPE_INTDIV      1
#define FPE_INTOVF      2
#define FPE_FLTDIV      3
#define FPE_FLTOVF      4
#define FPE_FLTUND      5
#define FPE_FLTRES      6
#define FPE_FLTINV      7
#define FPE_FLTSUB      8
#define NSIGFPE         8

#define SEGV_MAPERR     1
#define SEGV_ACCERR     2
#define NSIGSEGV        2

#define BUS_ADRALN      1
#define BUS_ADRERR      2
#define BUS_OBJERR      3
#define NSIGBUS         3
```

Figure 6-54: <sys/siginfo.h>, **Part 2 of 3**

```
#define TRAP_BRKPT      1
#define TRAP_TRACE      2
#define NSIGTRAP        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 NSIGCLD         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 NSIGPOLL        6

#define SI_MAXSZ        128
#define SI_PAD          ((SI_MAXSZ/sizeof(int))-3)

#define si_pad          _data._proc._pid
#define si_status       _data._proc._pdata._cld._status
#define si_stime        _data._proc._pdata._cld._stime
#define si_utime        _data._proc._pdata._cld._utime
#define si_uid          _data._proc._pdata._kill._uid
#define si_addr         _data._fault._addr
#define si_fd           _data._file._fd
#define si_band         _data._file._band
```

Figure 6-55: `<sys/siginfo.h>`**, Part 3 of 3**

```
typedef struct siginfo {
        int     si_signo;
        int     si_errno;
        int     si_code;
        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;
                }
                struct {
                        int     _fd;
                        long    _band;
                } _file;
        } _data;
} siginfo_t;
```

Figure 6-56: `<signal.h>`, **Part 1 of 2**

```
#define SIGHUP          1
#define SIGINT          2
#define SIGQUIT         3
#define SIGILL          4
#define SIGTRAP         5
#define SIGIOT          6
#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 SIGCLD          18
#define SIGCHLD         18
#define SIGPWR          19
#define SIGWINCH        20
#define SIGPOLL         22
#define SIGSTOP         23
#define SIGTSTP         24
#define SIGCONT         25
#define SIGTTIN         26
#define SIGTTOU         27
#define SIGURG          33
#define SIGIO           34
#define SIGXCPU         35
#define SIGXFSZ         36
#define SIGVTALRM       37
#define SIGPROF         38
#define SIGLOST         40
```

Figure 6-57: `<signal.h>`, **Part 2 of 2**

```
#define NSIG            65
#define MAXSIG          64

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

#define SS_ONSTACK      0x00000001
#define SS_DISABLE      0x00000002

struct sigaltstack {
        char    *ss_sp;
        int     ss_size;
        int     ss_flags;
};
typedef struct sigaltstack stack_t;
typedef struct sigset {
        unsigned long s[2];
} sigset_t;

#define SIGNO_MASK      0xFF
#define SIGDEFER        0x100
#define SIGHOLD         0x200
#define SIGRELSE        0x400
#define SIGIGNORE       0x800
#define SIGPAUSE        0x1000
```

Figure 6-58: <sys/stat.h>, **Part 1 of 2**

```
#define ST_FSTYPSZ     16

struct  stat {
        dev_t           st_dev;
        ino_t           st_ino;
        mode_t          st_mode;
        nlink_t         st_nlink;
        uid_t           st_uid;
        gid_t           st_gid;
        dev_t           st_rdev;
        off_t           st_size;
        time_t          st_atime;
        unsigned long   st_ausec;
        time_t          st_mtime;
        unsigned long   st_musec;
        time_t          st_ctime;
        unsigned long   st_cusec;
        timestruc_t     st_atim;
        timestruc_t     st_mtim;
        timestruc_t     st_ctim;
        long            st_blksize;
        long            st_blocks;
        char            st_fstype[ST_FSTYPSZ];
        char            st_padding[408];
};
```

Figure 6-59: `<sys/stat.h>`**, Part 2 of 2**

```
#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 6-60: <sys/statvfs.h>

```
#define FSTYPSZ         16

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

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

Figure 6-61: <stddef.h>

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

Figure 6-62: <stdio.h>

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

#define NULL           0

#define BUFSIZ         1024
#define EOF            (-1)

#define stdin          (&__stdinb)
#define stdout         (&__stdoutb)
#define stderr         (&__stderrb)

extern  FILE           __stdinb;
extern  FILE           __stdoutb;
extern  FILE           __stderrb;
#define getchar()      getc(stdin)
#define putchar(x)     putc((x),stdout)

#define SEEK_SET       0
#define SEEK_CUR       1
#define SEEK_END       2
#define L_ctermid      9
#define L_cuserid      9
#define P_tmpdir       "/var/tmp/"
#define L_tmpnam       (sizeof(P_tmpdir) + 15)
```

Figure 6-63: <stdlib.h>

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

typedef struct {
        long int        quot;
        long int        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 6-64: `<stropts.h>`, **Part 1 of 4**

```
#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 6-65: `<stropts.h>`**, Part 2 of 4**

```
#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_RECVFD        (STR|016)
#define I_PEEK          (STR|017)
#define I_FDINSERT      (STR|020)
#define I_SENDFD        (STR|021)
#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 6-66: `<stropts.h>`, **Part 3 of 4**

```
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 6-67: <stropts.h>**, Part 4 of 4**

```
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

#define FMNAMESZ        8

struct bandinfo {
        unsigned char bi_pri;
        int           bi_flag;
};
```

Figure 6-68: `<termios.h>`, **Part 1 of 6**

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

#define IBSHIFT         8
#undef  _POSIX_VDISABLE

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
```

Figure 6-69: `<termios.h>`**, Part 2 of 6**

```
#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
#define IMAXBEL         0020000
```

Figure 6-70: <termios.h>, **Part 3 of 6**

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

Figure 6-71: `<termios.h>`**, Part 4 of 6**

```
#define CBAUD           077600000
#define B0              0
#define B50             00200000
#define B75             00400000
#define B110            00600000
#define B134            01000000
#define B150            01200000
#define B200            01400000
#define B300            01600000
#define B600            02000000
#define B1200           02200000
#define B1800           02400000
#define B2400           02600000
#define B4800           03000000
#define B9600           03200000
#define B19200          03400000
#define EXTA            03400000
#define B38400          03600000
#define EXTB            03600000
#define CSIZE           00000060
#define CS5             0
#define CS6             0000020
#define CS7             0000040
#define CS8             0000060
#define CSTOPB          0000100
#define CREAD           0000200
#define PARENB          0000400
#define PARODD          0001000
#define HUPCL           0002000
#define CLOCAL          0004000
#define LOBLK           0010000
#define RCV1EN          0020000
#define XMT1EN          0040000
#define CIBAUD          037700000000
#define PAREXT          04000000
```

Figure 6-72: `<termios.h>`, **Part 5 of 6**

```
#define ISIG            0000001
#define ICANON          0000002
#define XCASE           0000004
#define ECHO            0000010
#define ECHOE           0000020
#define ECHOK           0000040
#define ECHONL          0000100
#define NOFLSH          0000200
#define TOSTOP          0000400
#define ECHOCTL         0001000
#define ECHOPRT         0002000
#define ECHOKE          0004000
#define FLUSHO          0020000
#define PENDIN          0040000
#define IEXTEN          0100000

#define IOCTYPE         0xff00
```

Figure 6-73: <termios.h>, **Part 6 of 6**

```
#define TIOC            ('T'<<8)
#define TCSANOW         (TIOC|14)
#define TCSADRAIN       (TIOC|15)
#define TCSAFLUSH       (TIOC|16)

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

struct termios {
        tcflag_t        c_iflag;
        tcflag_t        c_oflag;
        tcflag_t        c_cflag;
        tcflag_t        c_lflag;
        char            c_pad1;
        cc_t            c_cc[NCCS];
};
```

Figure 6-74: <sys/time.h>, **Part 1 of 2**

```
#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];

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

Figure 6-75: <sys/time.h>, **Part 2 of 2**

```
struct itimerval {
        struct timeval it_interval;
        struct timeval it_value;
};

#define ITIMER_REAL             0
#define ITIMER_VIRTUAL          1
#define ITIMER_PROF             2

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

Figure 6-76: <sys/times.h>

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

Figure 6-77: <sys/tiuser.h>, **Service Types**

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

Figure 6-78: <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 6-79: `<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 6-80: `<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 6-81: <sys/tiuser.h>, **Transport Interface Data Structures, 1 of 2**

```
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;
};

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

Figure 6-82: <sys/tiuser.h>, **Transport Interface Data Structures, 2 of 2**

```
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 6-83: <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 6-84: <sys/tiuser.h>, **Fields of Structures**

```
#define T_ADDR          0x00000001
#define T_OPT           0x00000002
#define T_UDATA         0x00000004
#define T_ALL           0x00000007
```

Figure 6-85: <sys/tiuser.h>, **Events Bitmasks**

```
#define T_LISTEN        0x00000001
#define T_CONNECT       0x00000002
#define T_DATA          0x00000004
#define T_EXDATA        0x00000008
#define T_DISCONNECT    0x00000010
#define T_ERROR         0x00000020
#define T_UDERR         0x00000040
#define T_ORDREL        0x00000080
#define T_EVENTS        0x000000ff
```

Figure 6-86: `<sys/tiuser.h>`, **Flags**

```
#define T_MORE          0x00000001
#define T_EXPEDITED     0x00000002
#define T_NEGOTIATE     0x00000004
#define T_CHECK         0x00000008
#define T_DEFAULT       0x00000010
#define T_SUCCESS       0x00000020
#define T_FAILURE       0x00000040
```

Figure 6-87: `<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;
```

Figure 6-88: <ucontext.h>

```
#include <sys/regset.h>

typedef struct {
        int             version;
        gregset_t       gregs;
} mcontext_t;

#define MCONTEXT_VERSION        1

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[201];
} ucontext_t;

#define GETCONTEXT      0
#define SETCONTEXT      1
```

Figure 6-89: <uio.h>

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

Figure 6-90: <ulimit.h>

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

Figure 6-91: <unistd.h>, **Part 1 of 3**

```
#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
#undef _POSIX_VDISABLE

#define _POSIX_VERSION        *
#define _XOPEN_VERSION        *

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

Figure 6-92: <unistd.h>, **Part 2 of 3**

```
#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_MAXUMEMV         12
#define _SC_MAXUPROC         13
#define _SC_MAXMSGSZ         14
#define _SC_NMSGHDRS         15
#define _SC_SHMMAXSZ         16
#define _SC_SHMMINSZ         17
#define _SC_SHMSEGS          18
#define _SC_NMSYSSEM         19
#define _SC_MAXSEMVL         20
#define _SC_NSEMMAP          21
#define _SC_NSEMMSL          22
#define _SC_NSHMMNI          23
#define _SC_ITIMER_VIRT      24
#define _SC_ITIMER_PROF      25
#define _SC_TIMER_GRAN       26
#define _SC_PHYSMEM          27
#define _SC_AVAILMEM         28
#define _SC_NICE             29
#define _SC_MEMCTL_UNIT      30
```

Figure 6-93: <unistd.h>, **Part 3 of 3**

```
#define _SC_SHMLBA              31
#define _SC_SVSTREAMS           32
#define _SC_CPUID               33
#define _SC_PASS_MAX            34
#define _SC_PAGESIZE            36
#define _SC_XOPEN_VERSION       37

#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 _PC_BLKSIZE             10

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

Figure 6-94: <utime.h>

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

Figure 6-95: <utsname.h>

```
#define SYS_NMLN        256

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

Figure 6-96: <wait.h>

```
#define WSTOPPED        0177
#define WCONTINUED      0010
#define WUNTRACED       0004
#define WNOHANG         0100
#define WNOWAIT         0200
#define WEXITED         0001
#define WTRAPPED        0002
#define WTRACED         WTRAPPED

#define WSTOPFLG        0177
#define WCONTFLG        0177777
#define WSIGMASK        0177

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

#define WCOREFLG        0200

#define WCOREDUMP(stat)         ((stat)&WCOREFLG)
#define WEXITSTATUS(s)          (((s)&0xff00)>>8)
#define WIFCONTINUED(stat)      (WWORD(stat)==WCONTFLG)
#define WIFEXITED(s)            (WTERMSIG(s)==0)
#define WIFSIGNALED(s)          (!WIFEXITED(s)&&!WIFSTOPPED(s))
#define WIFSTOPPED(s)           ((WTERMSIG(s)==0x7f) && (((s)&0x80)==0))
#define WSTOPSIG(s)             (WIFSTOPPED(s)?WEXITSTATUS(s):0)
#define WTERMSIG(s)             ((s)&0x7f)
```

INDEX

INDEX

Index

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

Z