AT&T

SYSTEM V
APPLICATION BINARY INTERFACE

Motorola 88000 Processor 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 0-13-877655-5

UNIX
PRESS
A Prentice Hall Title

Contents

1 INTRODUCTION

2 SOFTWARE INSTALLATION

3 LOW-LEVEL SYSTEM INFORMATION

4 OBJECT FILES

5 PROGRAM LOADING AND DYNAMIC LINKING

6 LIBRARIES

Figures and Tables

1. INTRODUCTION

1. INTRODUCTION

1 INTRODUCTION

Motorola 88000 Processor and the System V ABI

The **System V Application Binary Interface**, or **ABI**, defines a system interface for compiled application programs. Its purpose is to establish a standard binary interface for application programs on systems that implement 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 M88000 processor architecture. Together, these two specifications, the generic **System V ABI** and the **System V ABI Motorola 88000 Processor Supplement**, constitute a complete *System V Application Binary Interface* specification for systems that implement the architecture of the M88000 processor.

How to Use the Motorola 88000 Processor ABI Supplement

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

As with the **System V ABI**, this specification references other publicly-available reference documents, especially the **MC88100 User's Manual**. 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.
- 150 MB quarter-inch cartridge tape in QIC-150 format.

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

3. LOW-LEVEL SYSTEM INFORMATION

3. LOW-LEVEL SYSTEM INFORMATION

3 LOW-LEVEL SYSTEM INFORMATION

Machine Interface

Processor Architecture

The *MC88100 User's Manual* defines the processor architecture. Programs intended to execute directly on the processor use the instruction set, instruction encodings, and instruction semantics of the architecture, with the following exceptions:

- A program shall use only the instructions defined by the architecture.
- A program shall execute neither an `xmem` nor an `lda` instruction with an immediate `IMM16` field.
- A program shall not rely on the occurrence of a trap upon execution of a `div` or `divu` instruction with a zero divisor.

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 M88000 architecture as a subset, providing additional instructions or capabilities. Programs that use those capabilities explicitly do not conform to the M88000 ABI. Executing those programs on machines without the additional capabilities gives undefined behavior.

Data Representation

Byte Ordering

ABI compliant programs shall use Big-Endian byte order in all interfaces described in this document. ABI compliant programs can assume that the Processor Status Register (PSR) byte order (BO) bit specifies Big-Endian byte order.

C Fundamental Types

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

Figure 3-1: C Scalar Types

Type	C	`sizeof`	Alignment (bytes)	MC88100
Integral	`signed char` `char`	1	1	signed byte
	`unsigned char`	1	1	unsigned byte
	`short` `signed short`	2	2	signed halfword
	`unsigned short`	2	2	unsigned halfword

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

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

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

CAUTION The `long double` type has the same size and alignment as the `double` type for this version of the ABI. This relationship is Level 2: future versions of the Motorola 88000 Processor ABI Supplement may provide a different `long double` type.

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.

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

Figure 3-2: Structure Smaller Than a Word

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

Byte aligned, `sizeof` is 1

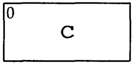


Figure 3-3: No Padding

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

Word aligned, `sizeof` is 8

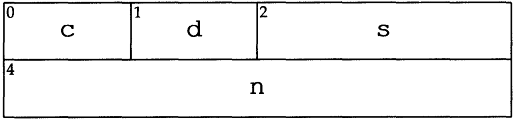


Figure 3-4: Internal Padding

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

Halfword aligned, `sizeof` is 4

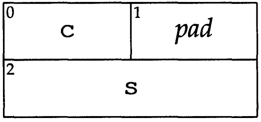


Figure 3-5: Internal and Tail Padding

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

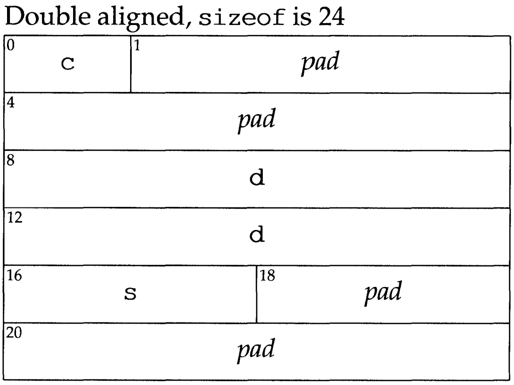
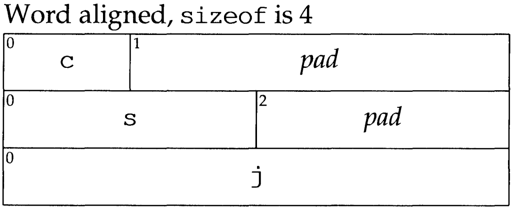


Figure 3-6: `union` **Allocation**

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



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 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

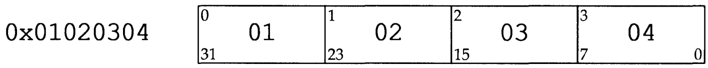


Figure 3-9: Left-to-Right Allocation

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

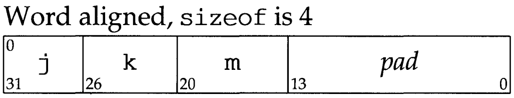


Figure 3-10: Boundary Alignment

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

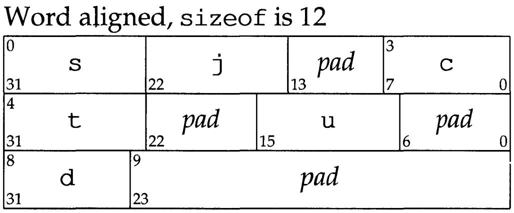


Figure 3-11: Storage Unit Sharing

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

Halfword aligned, `sizeof` is 2

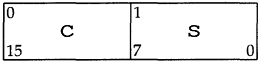


Figure 3-12: `union` Allocation

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

Halfword aligned, `sizeof` is 2

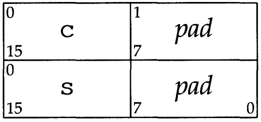
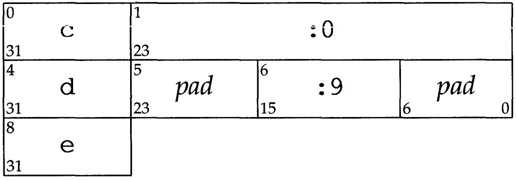


Figure 3-13: Unnamed Bit-Fields

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

Byte aligned, `sizeof` is 9



FORTRAN Data Types

Figure 3-14 shows the correspondence between FORTRAN's scalar types and the processor's.

Figure 3-14: FORTRAN Scalar Types

Type	FORTRAN	Size	Alignment (bytes)	MC88100
Character	`CHARACTER*(n)`	*n*	1	byte sequence
Integral	`LOGICAL`	4	4	word
	`INTEGER`	4	4	signed word
Floating-point	`REAL`	4	4	single-precision
	`DOUBLE PRECISION`	8	8	double-precision
	`COMPLEX`	8	4	paired single-precision

The `LOGICAL` data type has value `.FALSE.` if, and only if, it is binary zero. Otherwise, the value is `.TRUE.`.

Some FORTRAN programs that conform to ANSI Standard X3.9-1978 are not supported within this standard. Programs that force the compiler to produce misaligned storage allocation of double-precision real (typically using the `COMMON` and/or `EQUIVALENCE` statements) are not supported.

NOTE Support of these programs would degrade the performance of double-precision arithmetic in all programs. It is suggested that conforming compilers inform the user of such a misalignment.

Figure 3-15 shows additional, optional FORTRAN scalar types and their implementation on the MC88100.

Figure 3-15: Optional FORTRAN Scalar Types

Type	FORTRAN	Size	Alignment (bytes)	MC88100
Integral	`LOGICAL*1`	1	1	byte
	`LOGICAL*2`	2	2	halfword
	`LOGICAL*4`	4	4	word
	`INTEGER*1`	1	1	signed byte
	`INTEGER*2`	2	2	signed halfword
	`INTEGER*4`	4	4	signed word
Floating-point	`REAL*4`	4	4	single-precision
	`REAL*8`	8	8	double-precision
	`COMPLEX*8`	8	4	paired single-precision
	`COMPLEX*16` `DOUBLE COMPLEX`	16	8	paired double-precision

An array uses the same alignment as its elements.

NOTE The COMPLEX and COMPLEX*8 data types are 4 rather than 8 byte aligned as they are often equivalenced to two REAL data types.

COBOL Data Types

NOTE COBOL data types are defined not only to promote interlanguage operability but also to promote exchange of data with existing applications.

COBOL contains five categories of data items grouped into three classes. The *alphabetic class* contains the alphabetic category. The *numeric class* contains the numeric category. The *alphanumeric class* contains the numeric edited, alphanumeric edited, and alphanumeric categories.

- The alignment of the group is the maximum of the alignments of its elements.
- The elements of the group, in the order in which they appear in the source language, are assigned increasing positions, relative to the beginning of the group, in the structure representation. Each elementary item is assigned to the lowest available offset with the appropriate alignment. Note that this may require internal padding.
- A group's size is increased the minimum amount necessary (possibly zero) to make it an integral multiple of the group's alignment only if the group has an `OCCURS` clause. Note that this may require tail padding (only when there is an `OCCURS` clause).

Level 01 and 77 items alignment may use more restrictive alignment.

COBOL Standard Nonnumeric Data Types

All data types that belong to the alphabetic and alphanumeric classes are represented as a sequence of 8-bit ASCII characters, one character per byte, with byte alignment. The first, or leftmost, character at the COBOL source level is the lowest addressed byte of the representation.

COBOL Standard Numeric Data Types

The data types of the numeric class are, for the purposes of this standard, differentiated primarily by the `USAGE` and `SIGN` clauses of their COBOL source descriptions. The numeric data types described in the ANSI standard are `DISPLAY`, `PACKED-DECIMAL`, `BINARY`, and `COMPUTATIONAL`. `COMPUTATIONAL` shall use the same format as `BINARY`.

The implied decimal point in COBOL does not occupy a storage location. Numeric items described in terms of pseudo-PICTURE character strings with no implied decimal point represent all such numeric items without regard to the implied decimal point.

COBOL Standard Numeric Data Types — DISPLAY A numeric data item described, explicitly or implicitly, as USAGE IS DISPLAY is represented as one ASCII decimal digit character for each digit position (i.e., each 9) in the PICTURE for the item, aligned on a byte boundary. The high-order digit shall be the lowest addressed byte of the representation. The representation of ASCII decimal digits is:

Figure 3-16: COBOL ASCII Digits

Digit	Decimal	Hexadecimal
0	48	30
1	49	31
2	50	32
3	51	33
4	52	34
5	53	35
6	54	36
7	55	37
8	56	38
9	57	39

Unsigned data items shall contain one byte for each digit position.

Separate sign representations (SIGN IS LEADING/TRAILING SEPARATE) shall be the ASCII plus sign (+) for nonnegative numeric values and the ASCII minus sign (–) for negative numeric values. The representation of the data item shall contain one byte for each digit position plus one byte for the sign character. The sign character shall be the lowest (LEADING) or highest (TRAILING) addressed byte of the representation.

Combined sign representations (SIGN IS LEADING/TRAILING) shall combine the representation of the high-order (LEADING, most significant) or low-order (TRAILING, least significant) digit position with the operational sign for the item. A conforming implementation for COBOL shall be able to consume data using both combined sign representation variants shown in the following tables and shall document which variant(s) is (are) produced by that implementation.

Figure 3-17: COBOL Sign Representations, Part 1 of 2

	Nonnegative			Negative		
Digit	Decimal	Hex	ASCII	Decimal	Hex	ASCII
0	123	7B	{	125	7D	}
1	65	41	A	74	4A	J
2	66	42	B	75	4B	K
3	67	43	C	76	4C	L
4	68	44	D	77	4D	M
5	69	45	E	78	4E	N
6	70	46	F	79	4F	O
7	71	47	G	80	50	P
8	72	48	H	81	51	Q
9	73	49	I	82	52	R

NOTE These combined sign representations allow the translation of numeric values with combined signs from/to EBCDIC files without knowledge of the location of numeric fields within a record area. While such a capability lies outside ANSI X3.23-1985, which specifies that SIGN IS SEPARATE is required when CODE SET is specified for a file, current practice dictates that the exchange of combined sign data is necessary.

Figure 3-18: COBOL Sign Representations, Part 2 of 2

	Nonnegative			Negative		
Digit	Decimal	Hex	ASCII	Decimal	Hex	ASCII
0	48	30	0	112	70	p
1	49	31	1	113	71	q
2	50	32	2	114	72	r
3	51	33	3	115	73	s
4	52	34	4	116	74	t
5	53	35	5	117	75	u
6	54	36	6	118	76	v
7	55	37	7	119	77	w
8	56	38	8	120	78	x
9	57	39	9	121	79	y

NOTE The ABI anticipates that data fields using both representations may exist within a single record. Interoperability is promoted by the ability to consume both representations.

COBOL Standard Numeric Data Types — `PACKED-DECIMAL` A numeric data item described explicitly as `USAGE IS PACKED-DECIMAL` is represented as one 4-bit binary coded decimal (BCD) digit for each digit position (i.e., each `9`) in the `PICTURE` for the item. Two BCD digits are placed in each byte, with the lowest order digit in the most significant four bits and the operational sign representation in the least significant four bits of the highest addressed byte. The high-order digit shall be contained in the lowest addressed byte of the representation; if an even number of digits is specified in the `PICTURE` for the item, the high-order digit shall be in the least significant four bits and the most significant four bits shall be zero. The item is aligned on a byte boundary. The digit representations are as follows:

Digit	Decimal	Hexadecimal
0	0	0
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
8	8	8
9	9	9

A conforming implementation for COBOL shall be able to consume data using both sign representation variants shown below and shall document which variant(s) is (are) produced by that implementation.

Figure 3-19: COBOL Sign Variants

	Nonnegative	Negative	Unsigned
Variant A	0xC	0xD	0xF
Variant B	0xF	0xD	0xF

NOTE The ABI anticipates that data fields using both representations may exist within a single record. Interoperability is promoted by the ability to consume both representations.

COBOL Standard Numeric Data Types — `BINARY` A numeric data item described explicitly as `USAGE IS BINARY` is represented as a 16-, 32-, or 64-bit binary integer depending on the number of digit positions (i.e., 9's) in the `PICTURE` for the item.

If the item is signed (the `PICTURE` character string contains an S) the binary representation is a 2's complement binary integer. The sign bit shall be the most significant bit of the lowest addressed byte of the binary integer. The remaining seven bits of the lowest addressed byte shall contain the most significant portion of the binary integer and the highest addressed byte shall contain the least significant portion of the binary integer.

If the item has no sign, the binary representation is an unsigned binary integer. The lowest addressed byte shall contain the most significant portion of the binary integer and the highest addressed byte shall contain the least significant portion of the binary integer.

As permitted by Sections 5.13.4 (9) and 5.14.4 (3) of ANSI X3.23-1985, the alignment and size of `BINARY` data items shall be as specified in the following table:

Figure 3-20: COBOL `BINARY` Alignments

Digit Positions	Size	Alignment
1-4	16-bit	2 byte
5-9	32-bit	4 byte
10-18	64-bit	4 byte

Binary representations of numbers that cannot be specified in the number of decimal digits coded in the `PICTURE` for the item are nonstandard.

COBOL Nonstandard Numeric Data Types

Floating-point data types are not part of ANSI Standard COBOL and, therefore, are an optional part of this standard. A conforming implementation of COBOL shall adhere to ANSI/IEEE Std 754-1985 when providing these data types.

Function Calling Sequence

This section discusses the standard function calling sequence, including stack frame layout, register usage, parameter passing, etc. C, FORTRAN, and COBOL programs and their libraries use this calling sequence. The system libraries described in Chapter 6 require this calling sequence.

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

Registers and the Stack Frame

The MC88100 provides 32 general purpose registers, each 32 bits wide. Brief register descriptions appear in Figure 3-21.

Figure 3-21: Processor Registers

Register Name	Usage
`#r0`	Always equal to zero
`#r1`	Holds the subroutine return pointer
`#r2` to `#r9`	Temporary register set used for parameter passing
`#r10` to `#r13`	Temporary registers used for language-specific purposes
`#r14` to `#r25`	Preserved registers
`#r26` and `#r27`	Temporary registers
`#r28` and `#r29`	Reserved for ABI future use
`#r30`	Preserved register
`#r31`	Contains the stack pointer

Some registers have assigned roles.

`#r0`	Register `#r0` contains the constant zero.
`#r1`	Register `#r1` contains the return pointer generated by `bsr` or `jsr` instructions. Register `#r1` may be destroyed across subroutine calls.
`#r2` through `#r9`	This set of registers may be modified across procedure invocations and shall therefore be presumed by the calling procedure to be destroyed. These temporary registers are used for passing parameters to the called procedure.
`#r10` through `#r13`	Registers `#r10` through `#r13` are also used as temporary registers. These registers may be destroyed across subroutine calls. Registers `#r11`, `#r12`, and `#r13` have been allocated for some specific language requirements. Register `#r11` is used to pass the environment to a dummy procedure in FORTRAN. Register `#r11` is also used as a scratch register by the dynamic linking mechanism. See Chapter 5 for details. Register `#r12` is used by a calling procedure to pass an address to a called procedure when the calling procedure expects a result to be stored in an area of memory. The called procedure shall return its result in this area pointed to by the value in `#r12`, while the size in bytes is passed in `#r13`, if required by the language.
`#r14` through `#r25`	This set of registers shall be saved by the called procedure. They are used when values must be preserved for the duration of the current routine.
`#r26` and `#r27`	This set of registers may be modified across procedure invocations and shall therefore be presumed by the calling procedure to be destroyed.
`#r28` and `#r29`	A conforming program shall neither change nor rely on the contents of these registers.
`#r30`	Register `#r30` is a preserved register and shall be saved by the called procedure.

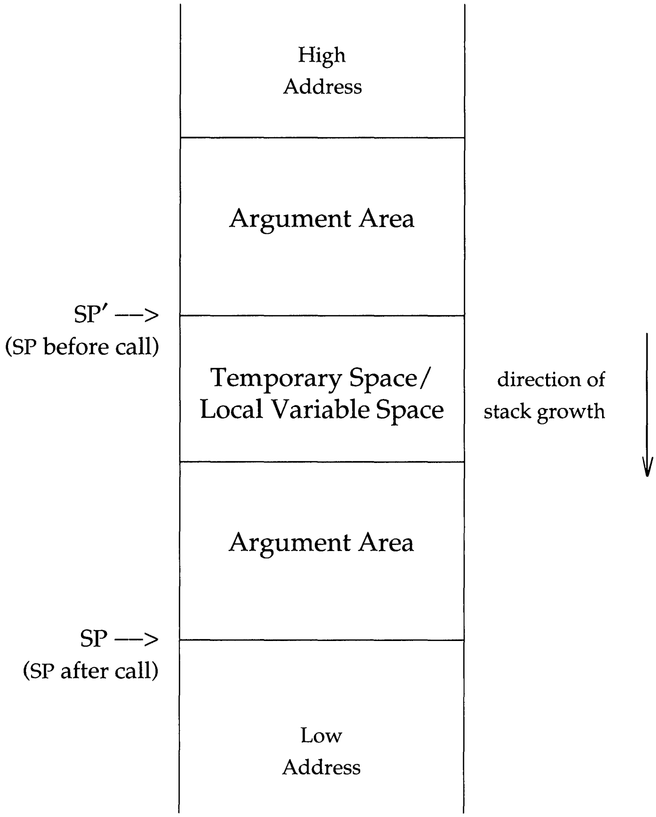
`#r31`	The stack pointer (stored in `#r31`) shall maintain 16-byte alignment. It shall point to the last word allocated on the stack, and grow towards low addresses. If required, it shall be decremented by the called procedure and incremented prior to returning.

Registers `#r14` through `#r25` and `#r30`, which are visible to both a calling and a called function, ''belong'' to the calling function. In other words, a called function shall save these registers' values before it changes them, restoring their values before it returns. Registers `#r1` through `#r13`, `#r26`, and `#r27` ''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.

Signals can interrupt processes [see `signal`(BA_OS)]. Functions called during signal handling have no unusual restrictions on their use of registers. A compiler may generate code that causes programs to use any register without the danger of signal handlers inadvertently changing their values.

In addition to the registers, each function may have a frame on the run-time stack. This stack grows downward from high addresses. Figure 3-22 shows the stack frame organization.

Figure 3-22: Stack Organization



SP denotes the stack pointer of the called subroutine at entry while SP′ denotes the stack pointer of the calling subroutine at entry.

Several key points about the stack frame deserve mention.

- The stack pointer shall maintain 16-byte alignment.
- The stack pointer shall point to the last word allocated on the stack and shall grow towards low addresses.
- The stack pointer shall be decremented by the called procedure on entry, if required, and incremented prior to return.
- 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 "local variable space" of the standard stack frame.
- The argument area shall be allocated by the caller and shall be at least 32 bytes. Its contents are not preserved across calls.
- The presence of the temporary space/local variable space depends on the nature of the function.

Across function boundaries, the function prologue may consist of several operations that depend on the nature of the function. Stack space may be allocated if the function:

- Uses the preserved registers and therefore must save and restore them
- Calls another function and therefore must save `#r1`, allocate the argument area, and possibly save any parameters.
- Needs local variables or temporary space.

The standard function prologue performs any or all of the following tasks, as needed:

- Allocates stack space
- Saves `#r1`
- Saves the address of the memory return value passed in `#r12`
- Saves parameters passed in registers `#r2-#r9`
- Saves registers `#r14` through `#r25`

Figure 3-23 illustrates an example of the function prologue allocating 88 bytes for local storage, and saving registers `#r24` and `#r25`. Eighty bytes are for local storage, and an additional 8 bytes are used for saving registers `#r24` and `#r25`.

Figure 3-23: Function Prologue

```
fcn:
        subu    #r31,#r31,96
        st.d    #r24,#r31,88
```

The standard function epilogue performs the following tasks, as needed:

- Either loads the return value or copies the result to the area pointed to by the pointer received in `#r12`
- Restores registers `#r14` through `#r25` and `#r30`
- Deallocates local stack space

If the function returns no value, or if the return register(s) already contain(s) the desired value, and no local stack was allocated, the epilogue in Figure 3-24 would suffice.

Figure 3-24: Simple Function Epilogue

```
fcnend:
            jmp   #r1
```

For a function that uses register `#r25`, is not a leaf function (i.e., may call another function and therefore may modify `#r1`), and requires a total of 80 bytes of local stack space, the following epilogue might be used:

Figure 3-25: Function Epilogue

```
fcnend:
          ld      #r25,#r31,72
          ld      #r1,#r31,76
          addu    #r31,#r31,80
          jmp     #r1
```

Argument Transmission

There is an offset in the argument area corresponding to each argument. The calling procedure shall use the offset as if all of the parameters were passed in memory with the first parameter at offset zero, and subsequent parameters passed consecutively. The offset is always rounded up to a multiple of 4 bytes. For arguments with greater than 4-byte alignment, the offset is always rounded up to a multiple of that alignment.

Arguments shall be at least word-aligned objects, and shall always be an integral number of words long. The first 8 words of the argument list will be passed in registers `#r2` through `#r9`, and not in the argument area. The first word of the argument list is passed in `#r2`, the second in `#r3`, etc., allocating registers consecutively until the eighth word is passed in `#r9`. The remainder of the argument list will be passed in memory, starting at an offset of 8 words from the start of the argument area.

The following subsections detail the mapping from the requirements of the specific language to the rules listed here, and also specify special cases that form exceptions to the rules stated here.

Argument Transmission for C

For the language C, signed short and characters are sign-extended to 32 bits before being passed. Unsigned short and characters are zero-extended to 32 bits before being passed. Any pointer, floating-point, integer, 4-byte aligned 4-byte structure, or 4-byte aligned 4-byte union argument whose offset is less than 32, is passed in the register numbered (or, for double-precision, the register pair beginning with the register numbered) 2+(*offset*/4).

All other arguments are passed at *offset* bytes from the beginning of the argument area.

Argument Transmission for FORTRAN

All actual arguments are passed by reference, i.e., a pointer to the argument is passed. Values transmitted in the argument area whose offset is less than 32 are passed in registers.

A procedure argument is represented by a 4-byte aligned instance of the following structure:

```
struct proc {int entry; int envir;}
```

where `entry` is the address of the first instruction of the procedure, and `envir` is the "environment" for the procedure.

For an actual argument that is a procedure, the address of a `proc` structure instance is passed. The `envir` member of this structure is unspecified; a value of zero is recommended.

When a dummy procedure is invoked, control is transferred to the address in the `entry` member of the associated `proc` structure instance. At time of transfer, register `#r11` contains the content of the `envir` member of the structure instance. Otherwise, the rules for dummy procedure invocation are the same as for external procedure invocation.

NOTE The representation of a procedure includes an environment in order to provide interoperability with languages that have internal procedures.

The FORTRAN character data type requires the passing of length as well as data address. In order to keep the other fundamental data types in compliance with the general rules outlined in the "Argument Transmission" section and to promote interoperability with other languages, FORTRAN establishes the length information for each string after passing all other arguments (including the character data addresses). The length in bytes of each character argument is passed as a 32-bit quantity at a position in the argument area based on the following formula:

given:	*arg1*, *arg2*, . . ., *arg*C, . . ., *arg*N as the actual argument list
where:	*arg*C is of type CHARACTER; there are N actual arguments total, and *arg*C is the Cth argument
then:	the length of *arg*C will be passed with offset 4N+4(C–1). If *arg*C is the last actual argument of type CHARACTER, the argument area shall be at least 4N+4C bytes in size. If *arg*X is the Xth actual argument and is not of type CHARACTER, the value at offset 4N+4(X–1) is undefined.

Argument Transmission for COBOL

The argument transmission for all data types is done by passing the address of the argument according to the convention outlined in the general rules of the "Argument Transmission" section.

Result Transmission

Results may be returned in registers or in memory. Registers `#r2` through `#r9` are available to return results. When results are returned in memory, the calling procedure allocates such memory and passes a pointer to it in `#r12`. The called procedure will then perform the copy to this area. If the language requires a size for this area, then the size in bytes shall be passed in `#r13`.

Other data types are returned by copying the return value to the memory area pointed to by the address contained in `#r12` at subroutine entry.

The following subsections detail the mapping from the requirements of each specific language to the rules specified in this section, and also specify special cases that form exceptions to the rules stated here.

Result Transmission for C

In the language C, single-precision floating-point, pointers, 4-byte aligned 4-byte structures, and 4-byte aligned 4-byte unions are returned in `#r2`. Signed integers and characters are sign-extended to 32 bits and returned in `#r2`. Unsigned integers and characters are zero-extended to 32 bits and returned in `#r2`. Double-precision floating-point values are returned in the register pair `#r2` and `#r3`. Other types are returned via memory as described in the general rules of "Result Transmission."

A function declared to return a `float` returns a single-precision value.

Result Transmission for FORTRAN

FORTRAN follows the general rules outlined in "Result Transmission" with the following additions.

`INTEGER` variant data types of size less than 4 bytes are sign-extended to 4 bytes before being returned. `LOGICAL` variant data types of size less than 4 bytes are extended to 4 bytes before being returned.

One word results are returned in register `#r2`. `DOUBLE PRECISION` and `REAL*8` results are returned in registers `#r2` and `#r3`.

`COMPLEX`, `COMPLEX*8`, `COMPLEX*16`, and `DOUBLE COMPLEX` functions return their result by placing the data in memory at the location addressed by register `#r12` (on entry to the function). The value in register `#r13` (on entry to the function) is unused.

`CHARACTER` functions return their result by placing the data in memory at the location addressed by register `#r12` (on entry to the function) padded or truncated to the length in bytes of the data area given by register `#r13` (on entry to the function).

Calls to fixed-sized CHARACTER functions, as well as those to `CHARACTER*(*)` functions, pass the length in `#r13`.

NOTE This method does not interoperate with C structure returning functions except when the size of the structure is known to equal the value of `#r13`.

Result Transmission for COBOL

There are no value-returning functions in COBOL.

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

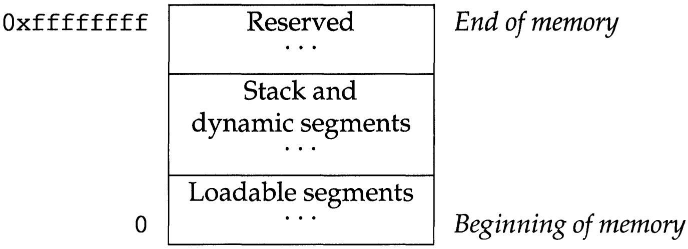
Memory is organized by pages, which are the system's smallest units of memory allocation. Page size can vary from one system to another. The allowable page sizes are 4K, 8K, 16K, 32K, or 64K. Processes can call `sysconf`(BA_OS) to determine the system's current page size.

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 3-26: Virtual Address Configuration



Loadable segments	Processes' loadable segments may begin at `0`. The exact addresses depend on the executable file format (see Chapters 4 and 5).
Stack and dynamic segments	A process's stack and dynamic segments reside below the reserved area. Processes can control the amount of virtual memory allotted for stack space, as described below.
Reserved	A reserved area resides at the top of virtual space.

NOTE Although application programs may begin at virtual address 0, they conventionally begin above `0x10000` (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 M88000 architecture: user and supervisor. Processes run in user mode (the less privileged). The operating system kernel runs in supervisor mode. A program executes a 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 *MC88100 User's Manual* describes, instruction execution can generate exceptions. The operating system handles such an exception either by completing the faulting operation in a manner transparent to the application, or by delivering a signal to the application. The correspondence between exceptions and signals is given in Figures 3-27 and 3-28.

The signals that an exception may give rise to are `SIGSEGV`, `SIGILL`, `SIGBUS`, `SIGTRAP`, and `SIGFPE`. If one of these signals is generated due to an exception when the signal is blocked, the behavior is undefined.

Due to the pipelined nature of the MC88100, more than one instruction may be executing concurrently. When an exception occurs, the operating system causes all executing instructions to complete their executions. As a result of completing these executions, additional exceptions may be generated. At most one of these concurrent exceptions is a precise exception; all the others are necessarily imprecise.

The operating system partitions the set of concurrent exceptions into subsets, all of whose exceptions share the same signal number. Each subset of exceptions is delivered as a single signal. The multiple signals resulting from multiple concurrent exceptions are delivered in unspecified order, except that, if there is a precise exception among the concurrent exceptions, the signal corresponding to the precise exception shall be delivered first.

When a signal representing an exception is delivered and the extended signal handler interface is selected with the `SA_SIGINFO` `sigaction(BA_OS)` flag, the information communicated through the second and third arguments is as follows. In the `siginfo` structure, `si_signo` contains the signal number; `si_machinexcep` contains the value 1; `_ncodes` contains the number of concurrent exceptions associated with this signal; `_exblks` points to an array of `exblk_t` structures consisting of `_ncodes` elements; and `si_code` contains a code identifying the cause of the signal. In each of the `exblk_t` elements, `eb_signo` contains the signal number; `eb_code` contains the code for the particular kind of exception, as indicated in Figures 3-27 and 3-28; and the `_eb_registers` union contains additional information about the exception, as indicated in Figures 3-27 and 3-28. In the `mcontext_t` structure of the `ucontext_t` structure, `version` contains the value 1; and the `gregs` array contains values for the indicated registers at the point of the exception. The effects of an instruction in progress at the time of the exception, including changes to registers and memory, are reflected in the machine state if and only if the given instruction completed successfully. For a precise exception, the value of the `R_XIP` element, with its low two bits cleared, locates the instruction generating the exception.

When a signal not representing an exception is delivered and the extended signal handler interface is selected with the `SA_SIGINFO` `sigaction` flag, the `si_machinexcep` member of the `siginfo` structure has the value 0.

Return from a signal handler handling a signal corresponding to an exception is permitted. The process state for resumption is that contained in the `ucontext_t` structure. In particular, the machine state for resumption is that contained in the (possibly modified) `gregs` array of the `mcontext_t` structure. Note that process execution is resumed at the addresses specified by the `R_NIP` and `R_FIP` values; the `R_XIP` value is ignored. Note also that the low two bits of the aforementioned register values are interpreted on resumption. See the *MC88100 User's Manual* for details.

Figures 3-27 and 3-28 show the relationship between machine exceptions and signals. The "Exception" column indicates the machine exception; see the *MC88100 User's Manual* for more details. The "P/I" column indicates whether the exception is precise ("P") or imprecise ("I"); see the *MC88100 User's Manual* for more details. The "Signal" column indicates the signal number under which the exception is delivered, if it is delivered. The "eb_code" column indicates the value assigned to the `eb_code` member of the `exblk_t` structure for the exception, when the `siginfo` structure is passed to the signal handling function. The `eb_registers` column indicates which member of the `_eb_registers` union is present, if any, when the `siginfo` structure is passed to the signal handling function.

Figure 3-27: Exceptions and Signals, Part 1 of 2

Exception	P/I	Signal	eb_code	_eb_registers	See Notes
Code access	P	`SIGSEGV`	`SEGV_CODE`	–	1
Data access	I	`SIGSEGV`	`SEGV_DATA`	`dfltinfo`	2,3
Misaligned data access	P	`SIGBUS`	`BUS_ALIGN`	–	4
Protection violation	I	`SIGBUS`	`BUS_PROT`	`dfltinfo`	3
Unimplemented opcode	P	`SIGILL`	`ILL_ILLOPC`	–	5
Privileged instruction violation	P	`SIGILL`	`ILL_PRVOPC`	–	
Integer overflow	P	`SIGFPE`	`FPE_INTOVF`	–	

Integer divide	P	`SIGFPE`	`FPE_INTDIV`
			`or`
			`FPE_INTOVF`
				–	6

Bounds check trap	P	`SIGFPE`	`FPE_FLTSUB`	–	
Trap to vectors 504-511	P	`SIGTRAP`	*vector number*	–	

Figure 3-28: Exceptions and Signals, Part 2 of 2

Exception	P/I	Signal	eb_code	_eb_registers	See Notes
Floating-point inexact	I	SIGFPE	FPE_FLTRES	fpifltinfo	7,8
Floating-point overflow	I	SIGFPE	FPE_FLTOVF or FPE_FLTRES	fpifltinfo	8,9
Floating-point underflow	I	SIGFPE	FPE_FLTUND or FPE_FLTRES	fpifltinfo	8,10
Floating-point divide by zero	P	SIGFPE	FPE_FLTDIV or FPE_FLTINV	-	11
Floating-point reserved operand	P	SIGFPE	FPE_FLTINV or FPE_FLTNAN	-	12
Floating-point integer conversion overflow	P	SIGFPE	FPE_FLTINV	-	13
Floating-point privilege violation	P	SIGFPE	FPE_PRIVVIO	-	
Floating-point unimplemented opcode	P	SIGFPE	FPE_UNIMPL	-	

Notes:

1. Code access exceptions caused by demand paging within the text segment and areas made executable [as by `mprotect`(KE_OS)] are handled transparently to the application.
2. Data access exceptions caused by references to the stack segment shall be handled by extending the stack in a manner transparent to the application, within the stack limits specified by `setrlimit`(BA_OS). Data access

exceptions caused by demand paging shall be handled transparently to the application.

3. The values of the members of `struct dfltinfo`, passed as `_eb_registers`, are the MC88100's Address Register, Transaction Register, and Data Register, respectively, of the memory transaction that caused the fault.
4. This exception can be disabled by setting the MXM bit of the Processor Status Register. (See the `setpsr()` function in ''Support Routines'' in Chapter 6.)
5. Conforming applications shall not use unimplemented opcodes.
6. If the faulting instruction is `div`, the dividend is the most negative integer and the divisor is –1, then the SIGFPE signal shall be sent with FPE_INTOVF as `eb_code`. If the divisor is zero for any integer division instruction, the SIGFPE signal shall be sent with FPE_INTDIV as `eb_code`. Otherwise, the faulting instruction must be `div` and one or both operands negative. In this case, the system completes the operation in a manner transparent to the application.
7. This exception can be disabled by clearing bit 0 (EFINX) of the Floating Point Control Register.
8. The values of the members of `struct fpifltinfo`, passed as `_eb_registers`, are the MC88100's Floating Point Result High Register, Result Low Register, and Imprecise Operation Type Register, respectively.
9. If bit 1 (EFOVF) of the FPCR is set, the SIGFPE signal shall be sent with FPE_FLTOVF as `eb_code`. Otherwise, bit 1 (AFOVF) of the FPSR shall be set, and if bit 0 (EFINX) of the FPCR is set, the SIGFPE signal shall be sent with FPE_FLTINEX as `eb_code`. If bit 0 of the FPCR is also clear, then bit 0 (AFINX) of the FPSR shall be set and the system shall complete the operation in a manner transparent to the application and consistent with ANSI/IEEE Std 754-1985 and the *MC88100 User's Manual*.
10. If bit 2 (EFUNF) of the FPCR is set, the SIGFPE signal shall be sent with FPE_FLTUND as `eb_code`. Otherwise, if there has been a loss of accuracy, bit 2 (AFUNF) of the FPSR shall be set. In this case, if bit 0 (EFINX) of the FPCR is set, the SIGFPE signal shall be sent with FPE_FLTINEX as `eb_code`; if it is clear, then bit 0 (AFINX) of the FPSR shall be set. If no signal is sent, the system shall complete the operation in a manner transparent to the application and consistent with ANSI/IEEE Std 754-1985 and the *MC88100 User's Manual*.

11. If the numerator is zero, the exception shall be handled as a floating-point reserved operand exception. Otherwise, if bit 3 (EFDVZ) of the FPCR is set, the `SIGFPE` signal shall be sent with `FPE_FLTDIV` as `eb_code`. If bit 3 of the FPCR is clear, then the system shall set bit 3 (AFDVZ) of the FPSR and complete the operation in a manner transparent to the application and consistent with ANSI/IEEE Std 754-1985 and the *MC88100 User's Manual*.

12. If the operation is the subtraction of two infinities, the multiplication of infinity and zero, or the division of one infinity by another, and bit 4 (EFINV) of the FPCR is set, then the `SIGFPE` signal shall be sent with `FPE_FLTOPERR` as `eb_code`; otherwise bit 4 (AFINV) of the FPSR shall be set. If either operand is a signaling NaN and bit 4 of the FPCR is set, then the `SIGFPE` signal shall be sent with `FPE_FLTNAN` as `eb_code`; otherwise bit 4 of the FPSR shall be set. If no signal is sent, the system shall complete the operation in a manner transparent to the application and consistent with ANSI/IEEE Std 754-1985 and the *MC88100 User's Manual*.

13. If the operand can be converted to an integer without overflow, the system shall complete the operation in a manner transparent to the application. If it cannot, and bit 4 (EFINV) of the FPCR is set, then the `SIGFPE` signal shall be sent. If bit 4 of the FPCR is clear, then bit 4 (AFINV) of the FPSR shall be set and the system shall complete the operation in a manner transparent to the application and consistent with ANSI/IEEE Std 754-1985.

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 3-29: Declaration for `main`

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

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

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

Registers

When a process is first entered (from an `exec()` system call), registers are initialized as follows:

`#r1`	is implementation-defined.
`#r2`	contains *argc*, the number of arguments.
`#r3`	contains *argv*, a pointer to the array of argument pointers in the stack. The array is immediately followed by a **NULL** pointer. If there are no arguments, `#r3` shall point to a **NULL** pointer.
`#r4`	contains *envp*, a pointer to the array of environment pointers in the stack. The array is immediately followed by a **NULL** pointer. If no environment exists, `#r4` shall point to a **NULL** pointer.
`#r5`	contains a pointer to the auxiliary vector. The auxiliary vector shall have at least one member, a terminating entry with an `a_type` of AT_NULL.
`#r6`	possibly contains a termination function pointer. If `#r6` contains a nonzero value, the value represents a function pointer that the application should register with `atexit`(BA_OS). If `#r6` contains zero, no action is required.

`#r7-#r13`	are currently set to zero. Future versions of the system might use the registers to hold special values, so applications should not depend on these registers' values.
`#r14-#r30`	are unspecified.
`#r31`	is the initial stack pointer, aligned to an 16-byte boundary.
FPSR	is the floating-point user status register. This register is initially cleared.
FPCR	is the floating-point user control register. This register is set to *round to nearest* mode and all the user exception handlers are disabled. Individual processes may change the register contents if desired.
PSR	is the Processor Status Register; it contains 0x3f0, which corresponds to:

- user mode,
- Big-Endian byte ordering,
- concurrent operation allowed,
- carry bit clear,
- SFU 1 enabled,
- SFU2-SFU7 disabled,
- misaligned accesses cause an exception,
- interrupts enabled,
- shadow registers enabled.

Individual programs may need to manipulate the stacked data and register contents at startup before control passes to the main section of the program.

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.

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 3-30: Auxiliary Vector

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

Figure 3-31: Auxiliary Vector Types, a_type

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

AT_NULL	The auxiliary vector has no fixed length; instead the end of the table is indicated by placing AT_NULL into a_type.
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 the 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 shall also be present. See Chapter 5 in both the **System V ABI** and this 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 M88000.
`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.

Coding Examples

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

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

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

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 Execute Instruction Pointer (XIP). A XIP-relative branch or function call computes its destination address in terms of the current XIP, *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 XIP-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 unshared segments. 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 unshared segments are private for each process, the table entries can change—unlike shared segments, which multiple processes share.

However, there still remains the problem of addressing the global offset table and the procedure linkage table in a position-independent manner. The M88000 architecture lacks instructions to reference data or compute addresses with XIP-relative addresses. The most efficient method to reference locations in a shared object is with based addressing. In this scheme, the address of the shared object is computed at execution time and held in a register. The offset from this address to any location in the shared object is known by the link editor when it is building the shared object, and this offset can be efficiently encoded in instructions.

In order to allow it to lay out the shared object as efficiently as possible, the link editor is given the responsibility of choosing the location in the shared object whose address at execution time will serve as the *addressing base*. Code generated for a shared object refers to the addressing base only indirectly, through a variety of relocation types that deal with the addressing base. The link editor records its choice of addressing base with the `DT_88K_ADDRBASE` value. (See Chapter 5 for more information.) One natural choice for the position of the addressing base is the address of the global offset table.

Do not confuse the related terms ''base address'' and ''addressing base.'' The base address of an executable or shared object file, as defined by the System V ABI, is the *lowest* virtual address associated with the memory image of the program's object file. In similar terms, the addressing base is a *particular* virtual address associated with the memory image of the program's object file. The addressing base of a shared object may coincide with its base address, but it need not.

Assembly language examples below show the explicit notation needed for position-independent code. In the descriptions below, the construction ''difference between X and Y'' means the 32-bit modulus subtraction $X - Y$.

s`#got`	This expression denotes the address of a global offset table entry for symbol *s*.
p`#gotp`	This expression denotes the address of a global offset table procedure entry for the procedure named by symbol *p*.
p`#plt`	This expression denotes an address to which control can be transferred to invoke the procedure named by symbol *p*. This address is either the address of *p* or the address of a procedure linkage table entry for *p*.

s`#rel`	This expression denotes the difference between the value of the symbol *s* and the addressing base for the shared object containing the expression. This expression is valid only in a shared object.
s`#got_rel`	This expression denotes the difference between the address denoted by *s*`#got` and the addressing base for the shared object containing the expression. This expression is valid only in a shared object.
p`#gotp_rel`	This expression denotes the difference between the address denoted by *p*`#gotp` and the addressing base for the shared object containing the expression. This expression is valid only in a shared object.
p`#plt_rel`	This expression denotes the difference between the address denoted by *p*`#plt` and the addressing base for the shared object containing the expression. This expression is valid only in a shared object.
s`#abdiff`	This expression denotes the difference between the addressing base for the shared object containing the expression and the value of the symbol *s*. The value of the symbol *s* must represent an address in the shared object containing the expression. This expression is valid only in a shared object.

Position-Independent Function Prologue and Epilogue

This section describes the function prologue and epilogue for position-independent code. A position-independent function generally needs to establish its addressing base to afford access to its private data, in particular its global offset table entries. The addressing base is typically computed into a preserved register, such as `#r25`, so that its value will be preserved throughout the activation of the function.

NOTE As a reminder, this entire section contains examples. Using `#r25` 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.

The prologue for a position-independent function `name` that needs 96 bytes of stack space and uses register `#r25` to hold the addressing base might be as shown in Figure 3-32.

Figure 3-32: Position-Independent Function Prologue

```
name:   subu    #r31,#r31,96
        st      #r25,#r31,88
        st      #r1,#r31,92
        bsr.n   here
        or.u    #r25,#r0,#hi16(here#abdiff)
here:   or      #r25,#r25,#lo16(here#abdiff)
        addu    #r25,#r25,#r1
```

The epilogue for the position-independent function `name` described above might be as shown in Figure 3-33.

Figure 3-33: Position-Independent Function Epilogue

```
name:   ld      #r25,#r31,88
        ld      #r1,#r31,92
        addu    #r31,#r31,96
        jmp     #r1
```

Data Objects

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

In the M88000 architecture, only load and store instructions access memory. Because instructions cannot hold 32-bit addresses directly, a program normally computes an address into a register. Symbolic references in absolute code put the symbols' values—or absolute virtual addresses—into instructions.

Figure 3-34: Absolute Load and Store

```
        C                          Assembly
extern int src;         global    src, dst, ptr
extern int dst;
extern int *ptr;
ptr = &dst;             or.u      #r2,#r0,#hi16(dst)
                        or        #r2,#r2,#lo16(dst)
                        or.u      #r3,#r0,#hi16(ptr)
                        st        #r2,#r3,#lo16(ptr)

*ptr = src;             or.u      #r2,#r0,#hi16(src)
                        ld        #r2,#r2,#lo16(src)
                        or.u      #r3,#r0,#hi16(ptr)
                        ld        #r3,#r3,#lo16(ptr)
                        st        #r2,#r3,0
```

Position-independent instructions cannot contain absolute addresses. Instead, instructions that reference symbols hold the offsets of the symbols' global offset table entries relative to the addressing base for the shared object. Combining the offset of the global offset table entry with the addressing base in `#r25` gives the absolute address of the table entry holding the desired address.

Figure 3-35: Position-Independent Load and Store

C	Assembly
`extern int src;`	`global  src, dst, ptr`
`extern int dst;`	
`extern int *ptr;`	
`ptr = &dst;`	`or.u    #r2,#r0,#hi16(dst#got_rel)`
	`or      #r2,#r2,#lo16(dst#got_rel)`
	`ld      #r2,#r25,#r2`
	`or.u    #r3,#r0,#hi16(ptr#got_rel)`
	`or      #r3,#r3,#lo16(ptr#got_rel)`
	`ld      #r3,#r25,#r3`
	`st      #r2,#r3,0`
`*ptr = src;`	`or.u    #r2,#r0,#hi16(src#got_rel)`
	`or      #r2,#r2,#lo16(src#got_rel)`
	`ld      #r2,#r25,#r2`
	`ld      #r2,#r2,0`
	`or.u    #r3,#r0,#hi16(ptr#got_rel)`
	`or      #r3,#r3,#lo16(ptr#got_rel)`
	`ld      #r3,#r25,#r3`
	`ld      #r3,#r3,0`
	`st      #r2,#r3,0`

Function Calls

A function call is typically made with a `bsr` instruction. A `bsr` instruction has a self-relative branch displacement that can reach 128 megabytes in either direction. Hence, use of a `bsr` instruction to effect a call within an executable or shared object file limits the size of the executable or shared object file to 128 megabytes. A `bsr` instruction can also be used to effect a call between two different object files, without constraining the placement of the two object files in memory, because control generally passes from the `bsr` instruction, through an indirection sequence, to the desired destination. See "Procedure Linkage Table" in Chapter 5 for more information on the indirection sequence.

Figure 3-36: Absolute Direct Function Call

C

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

Assembly

```
global   function
bsr      function#plt
```

or:

```
global   function
bsr      function
```

Figure 3-36 shows two methods for effecting a call in absolute code. Note that the `#plt` suffix can be supplied or omitted. Supplying the `#plt` suffix is convenient if it is desirable to make absolute and position-independent function calls in the same way. Omitting the `#plt` suffix is convenient if it is desirable to make absolute function calls the way they have been made traditionally.

Supplying the `#plt` suffix does not necessarily result in the use of a procedure linkage table entry. If caller and callee are both in the executable file, for example, no PLT entry is needed. On the other hand, omitting the `#plt` suffix may result in the use of a PLT entry. If the link editor determines that the executable file is making reference to a function defined in a shared object, the link editor uses a PLT entry for the reference.

Figure 3-37: Position-Independent Direct Function Call

C

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

Assembly

```
global   function
bsr      function#plt
```

or:

```
global   function
or.u     #r1,#r0,#hi16(function#gotp_rel)
or       #r1,#r1,#lo16(function#gotp_rel)
ld       #r1,#r25,#r1
jsr      #r1
```

Figure 3-37 shows two methods for effecting a call in position-independent code. If a `bsr` instruction is used, the `#plt` suffix should be supplied. Without the `#plt` suffix, a reference in a shared object to an external function resolves not to a PLT entry in the shared object, but to the canonical address for the function. (See "Function Addresses" in Chapter 5 for more information.) Such resolution compromises the position independence of the shared object.

As the second alternative in Figure 3-37 shows, the indirection of the procedure linkage table entry may be avoided by making direct reference to the global offset table procedure entry for the function. The instruction sequence shown assumes that the addressing base is held in register `#r25`.

Other sequences for effecting a direct function call are possible. For example, in absolute code, the global offset table procedure entry could be loaded directly and used with a `jsr` instruction. In position-independent code, the global offset table procedure entry can be loaded more concisely as long as there are not too many global offset table entries.

Figure 3-38: Absolute Indirect Function Call

C

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




 (*ptr)();
```

Assembly

```
global   ptr, name

or.u     #r2,#r0,#hi16(name)
or       #r2,#r2,#lo16(name)
or.u     #r3,#r0,#hi16(ptr)
st       #r2,#r3,#lo16(ptr)

or.u     #r1,#r0,#hi16(ptr)
ld       #r1,#r1,#lo16(ptr)
jsr      #r1
```

Figure 3-39: Position-Independent Indirect Function Call

C

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







 (*ptr)();
```

Assembly

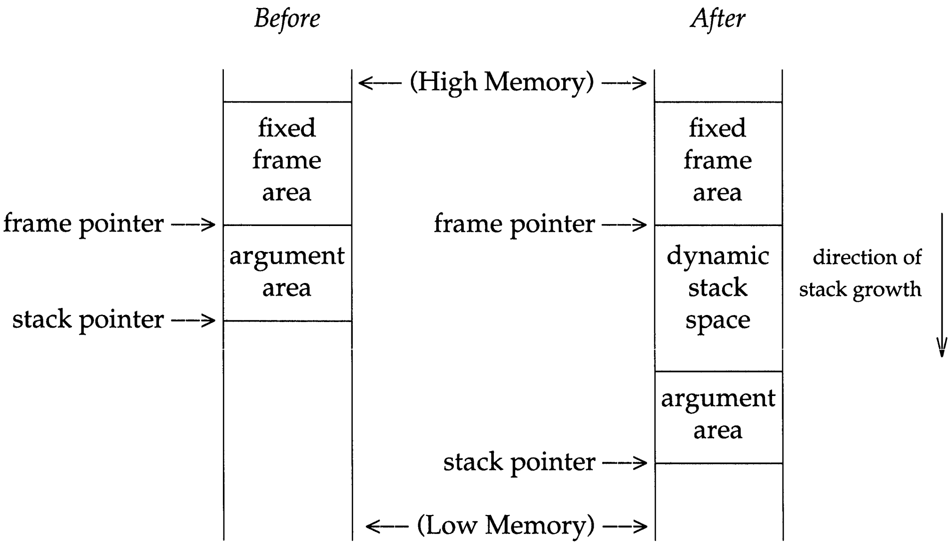
```
global   ptr, name

or.u     #r2,#r0,#hi16(name#got_rel)
or       #r2,#r2,#lo16(name#got_rel)
ld       #r2,#r25,#r2
or.u     #r3,#r0,#hi16(ptr#got_rel)
or       #r3,#r3,#lo16(ptr#got_rel)
ld       #r3,#r25,#r3
st       #r2,#r3,0

or.u     #r1,#r0,#hi16(ptr#got_rel)
or       #r1,#r1,#lo16(ptr#got_rel)
ld       #r1,#r25,#r1
ld       #r1,#r1,0
jsr      #r1
```

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. Portable C programs should use the header files `<stdarg.h>` or `<varargs.h>` to deal with variable argument lists (on MC88100 and other machines as well).

Allocating Stack Space Dynamically

Figure 3-40: Dynamic Stack Space Allocation



The M88000 architecture supports dynamic stack space allocation for those languages that require it. The mechanism for allocating dynamic space is embedded completely within a function and does not affect the standard calling sequence. Thus, functions that need dynamic stack frame sizes can call functions that do not, and vice versa.

A typical variant of the mechanism is described below and diagrammed in Figure 3-40. The figure shows the layout of a stack frame before and after dynamic stack allocation. The fixed frame area is used for storage of function data, such as local variables, whose sizes are known to the compiler. The fixed frame area is allocated at function entry and does not change in size or position during the function's activation. The argument area is used for storage of arguments passed in calls to other functions. Its size is also known to the compiler and can be allocated along with the fixed frame area at function entry. However, the standard calling sequence requires that the stack pointer locate the argument area, so the argument area must move when dynamic stack allocation occurs.

Data in the argument area are naturally addressed at constant offsets from the stack pointer. However, in the presence of dynamic stack allocation, the offsets from the stack pointer to the data in the fixed frame area are not constant. To provide addressability, a frame pointer is established to locate the fixed frame area consistently throughout the function's activation.

Dynamic stack allocation is accomplished by "opening" the stack just above the argument area. The following steps show the process in detail.

1. The amount of dynamic space to be allocated is rounded up to a multiple of 16 bytes, so that 16-byte stack alignment is maintained.
2. The stack pointer is decreased by the rounded byte count.
3. All active data in the argument area, if any, are copied from the previous position of the stack pointer to the new position. The amount of data to be copied is known to the compiler.
4. The address of the newly allocated dynamic stack space is the sum of the new value of the stack pointer and the size of argument area.

The above process can be repeated as many times as desired within a single function activation. When it is time to return, the stack pointer is first reset to its position as shown in the left portion of Figure 3-40, thereby removing all dynamically allocated stack space. Normal return processing may then ensue.

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

1. The signal handler can return. The process then resumes the dynamic allocation from the point of interruption.
2. 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.
3. The process can terminate.

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

Text Description Information

The M88000 ABI opts not to prescribe the form of a stack frame, in order to leave compilers with the greatest possible flexibility to generate efficient code. For example, no convention is defined to link stack frames at execution time, and a compiler may elect not to use a frame pointer for a particular routine. The lack of a traditional stack frame convention, however, would make low-level debugging impossible, were it not for the alternate convention described here.

This section defines a mechanism by which programs describe relevant aspects of their text sections. The essence of the mechanism is that information about important execution-time characteristics of procedures is provided statically, by the compiler and link editor, rather than dynamically, by executing instructions at runtime, wherever possible. Information describing a procedure is generated by the compiler and is associated with the procedure by the link editor. When the information relevant to a particular text address is needed, the text address is mapped to the procedure containing the address, and the text description information associated with the procedure is consulted.

Text description information describes code in an object file. This code is referred to as "text" because it usually resides in the `.text` section. However, code may reside in other sections with attributes similar to those of the `.text` section, and even in sections with attributes similar to those of the `.data` section, provided that the latter sections are made executable during execution. References to "text" should be taken to mean references to "code" in its more general form.

Tdesc Information

A *text chunk* is a contiguous sequence of zero or more words of text of an object file. A text chunk consisting of zero words is an *empty text chunk*. The *start address* of a non-empty text chunk is the minimum of the addresses of the words of the non-empty text chunk. The *end address* of a non-empty text chunk is the maximum of the addresses of the words of the non-empty text chunk, plus 4. The *start address* and *end address* of an empty text chunk are equal. The start address is inclusive and the end address is exclusive. An address is said to be "in" a text chunk if it is greater than or equal to the start address of the text chunk and less than the end address of the text chunk. A word is said to be "in" a text chunk if its address is in the text chunk.

Contributors of text (typically compilers and assemblers) shall partition that text into one or more text chunks. All text chunks so defined for an object file must not overlap; that is, no word may be in more than one text chunk.

Contributors of text identify a text chunk and associate information descriptive of that text chunk by contributing a "tdesc chunk" to the `.tdesc` section. The `.tdesc` section is system-defined. It has the `SHF_ALLOC` attribute, it does not have the `SHF_WRITE` attribute, and it may or may not have the `SHF_EXECINSTR` attribute.

A *tdesc chunk* begins on a word boundary and is a contiguous sequence of words with the following structure:

Figure 3-41: Tdesc Chunk

Word Position	Bit Range	Interpretation
0	31 - 24	zeroes
	23 - 2	info length, in bytes
	1 - 0	info alignment exponent
1	31 - 0	info protocol
2	31 - 0	start address of text chunk
3	31 - 0	end address of text chunk
4+		info

The zeroes in word 0 are designed to be distinct from the high 8 bits of typical M88000 "no-op" instructions (instructions that, when executed, have no effect). This allows possible padding between tdesc chunks to be detected, whether the padding consists of words of zeroes or no-op instructions.

The *info protocol* describes the form and interpretation of the tdesc chunk, primarily that of its info portion.

The info protocol represents a contract between compiler and debugger/runtime system. Providing for different info protocols allows different (through space and time) compilers to use different strategies for describing their code.

The *info length* is the number of bytes of meaningful information that begin in word 4 of the structure. The tdesc chunk is padded with 0 to 3 bytes of undefined information to make its total size an integral multiple of 4 bytes.

The *info alignment exponent* indicates the required alignment for the info field after the link editor has collected and reformatted the tdesc information (as described later). The *info alignment exponent* specifies the required alignment according to the following table:

Figure 3-42: Info Field Alignment

info alignment exponent	alignment in bytes
0	1
1	2
2	4
3	8

NOTE Alignments greater than 8 are not supported.

Info Protocol

Two info protocols are defined. They are identified with the integers 1 and 2. The only difference between the two protocols is the interpretation of the start address and end address of the text chunk. For protocol 1, the addresses are absolute; for protocol 2, the addresses are relative to the addressing base for the shared object containing the tdesc chunk. Hence, info protocol 2 can be used only in a shared object. Otherwise, the two info protocols are the same. For both protocols the info length is always 16 and the info alignment exponent is always 2. The structure of the info for both protocols is as follows:

Figure 3-43: Info Structure

Word Position	Bit Range	Interpretation
0	31 - 24	info variant, the integer 1
	23 - 7	register save mask, for registers `#r14-#r30`; bit 7 is the `#r30` save mask, bit 8 for `#r29`, etc., consecutively until bit 23 for `#r14`
	6	zero
	5	return address info discriminant
	4 - 0	frame address register
1	31 - 0	frame address offset
2	31 - 0	return address info
3	31 - 0	register save offset

The above structure is the only currently defined variant. Zeroes are required where no useful information is defined to facilitate future extension.

The *info* field of the tdesc chunk describes important low-level characteristics of the execution environment which is in effect when the instruction pointer is in the associated text chunk. Because the information in the tdesc chunk is unchanging, it must depend on the context. The context consists of a text address and the values that the registers available to user-level programs would have were control about to proceed to the instruction addressed by the text address. The text address portion of a context is called its *instruction pointer*.

The *canonical frame address* (abbreviated "CFA") for a procedure is the value of the stack pointer at entry to the procedure. The CFA shall be computable from the procedure's context and its text chunk's associated tdesc chunk's info field as follows:

CFA = contents_of(frame address register) + frame address offset

where "+" represents machine address arithmetic, and "contents_of(*register*)" represents the value of the indicated register in the procedure's context.

Procedures that construct a "frame pointer" in a register will specify that register as the frame address register and the difference between the initial stack pointer value and the contents of that register as the frame address offset. Procedures that do not construct a frame pointer explicitly will specify the stack pointer as the frame address register and the (necessarily) fixed frame size as the frame address offset.

The current size of a frame can be computed as follows:

frame size = CFA – contents_of(`#r31`)

where "–" represents mathematical subtraction. (The stack pointer is always housed in `#r31`.)

A *frame position* is a (byte) address relative to the CFA; that is, to calculate the address of a word at a frame position, sum (using machine address arithmetic) the CFA and the frame position. A frame position must be an integral multiple of 4. That is, frame positions mark word-aligned positions in the frame.

The *return address* for a procedure is the text address to which the procedure would return control were it to complete normally. Currently the return address must be "exact;" that is, a procedure is constrained to return exactly to the return address if it returns normally. The procedure housed in the text chunk that the return address of another procedure is in is known as the *parent* or *caller* of that other procedure. Note that, in the case of "tail call," the caller is not the procedure that passed control directly, but rather an ancestor of that procedure.

The return address shall be computable from the procedure's context and its text chunk's associated tdesc chunk's info field as follows: If the return address info discriminant is 0, the return address is the value of the register specified by the return address info field, with its low two bits cleared; if the return address info discriminant is 1, the return address is the value of the word at the frame position specified by the return address info field, with its low two bits cleared. A return address is always word-aligned. Ignoring the low two bits of return address values mimics the behavior of the hardware and allows other useful information to be stored there.

The return address for a procedure is contained in `#r1` at entry to the procedure. A procedure that calls another procedure must store the initial contents of `#r1` in its frame.

A *leaf procedure* (one that calls no other) may not need to store the contents of `#r1`, so its return address would remain there. However, a leaf procedure may need to free `#r1`, to use a `bsr` instruction which transfers within the procedure to locate the procedure in a position independent manner. In this case, the procedure may save the initial contents of `#r1` in another register instead of in its frame.

A return address value of zero indicates the absence of a parent text chunk and hence terminates a return address chain. The runtime initializer (typically `crt0`) shall have a return address, as described by its tdesc information, of zero. Stack traceback is achieved by following the chain of return addresses from callee to caller. A distinguished value for the end of this chain is required to make the traceback terminate.

The register save mask may have "1" bits only in bit positions corresponding to preserved register numbers. The register save mask must have a "1" bit in any bit position corresponding to a preserved register that is modified by the procedure. The values at procedure entry of the registers marked by "1" bits in the register save mask must be stored in the frame. The lowest-numbered register whose mask bit is "1" is stored at the frame position specified by the register save offset. Successively higher-numbered registers whose mask bits are "1" are stored in successive words in the frame at increasing addresses. A bit in the register save mask at position p, relative to the least significant bit of the mask, corresponds to the register numbered $30-p$.

Both the tdesc chunk header and info field for info protocols 1 and 2 are 16 bytes in length. This avoids padding with assemblers that pad section sizes to multiples of 16 bytes.

Typically, the execution environment of a procedure is not fully established until after several initial instructions have been executed. These initial instructions are often referred to as the procedure's *prologue*. Similarly, the procedure's execution environment is typically disestablished incrementally by final instructions referred to as the procedure's *epilogue*. That portion of a procedure which is neither prologue nor epilogue is termed *body*.

The simple information provided by a tdesc chunk with info protocols 1 and 2 can describe only a single, unchanging execution environment. This suffices for a single tdesc chunk to describe a procedure's body. However, the procedure's prologue and epilogue portions are not correctly described by the same tdesc information. Hence, the text chunk that covers the procedure's body must not also cover its prologue and epilogue sections. Additional text chunks can be defined to describe prologue and epilogue sections. However, because the execution environment typically changes frequently during prologue and epilogue sections, possibly many additional, small text chunks, each with its own tdesc chunk, would be required. For this reason, the requirement as to which instructions must be in a text chunk (and hence which must be described by tdesc information) is left purposely vague. Discretion is left to the implementation.

Map Protocol

The link editor treats tdesc information specially. When producing an executable file or shared object file, it reformats the contributions to the `.tdesc` section before making them part of a segment of the object file. The reformatted tdesc information may consist of one or more pieces. Each piece of tdesc information is aligned to a word boundary and has the following general structure:

Figure 3-44: Tdesc Information Piece

Word Position	Bit Range	Interpretation
0	31 - 0	map protocol
1+		info

The *map protocol* describes the form and interpretation of the *info* portion of the tdesc information piece.

The map protocol represents a contract between link editor and debugger/runtime system. Providing for different map protocols allows different (through space and time) link editors to use different strategies for mapping text addresses to tdesc chunks.

Two map protocols are defined. They are identified with the integers 1 and 2.

The structure of the *info* for map protocol 1 is shown in Figure 3-45.

Figure 3-45: Map Protocol 1

Word Position	Bit Range	Interpretation
0	31 - 0	end address of this structure
1+		tdesc chunk sequence

The first word of info gives the address just beyond the end of this piece of tdesc information. Beginning at the second word is a concatenation of all contributions to the `.tdesc` section, in arbitrary order. This concatenation includes all tdesc chunks and may include padding words before, between, and after tdesc chunks. A padding word is either a word all of whose bits are zero, or a word whose high 8 bits are not all zero. The required alignment of the *info* fields of the tdesc chunks shall be met. Hence, the only required "reformatting" performed for map protocol 1 is the addition of the map protocol word and the end address word. This map protocol is crude, but it is adequate to support debugging, because debugging performance is not critical.

The structure of the *info* for map protocol 2 is shown in Figure 3-46.

Figure 3-46: Map Protocol 2

Word Position	Bit Range	Interpretation
0	31 - 0	end address of this structure
1+		array of tdesc piece entries

The first word of info gives the address just beyond the end of this piece of tdesc information. The remainder of the piece is an array of structures with the following form:

Figure 3-47: Tdesc Piece Entry

Word Position	Bit Range	Interpretation
0	31 - 0	address of tdesc information piece
1	31 - 0	addressing base for piece

The first word gives the address of another piece of tdesc information. An address of zero represents an absent piece. If the first word is nonzero, the second word gives the addressing base for any immediately subordinate tdesc chunks with info protocol 2.

Together, map protocols 1 and 2 provide the capability to represent a tree of tdesc information. Map protocol 1 pieces serve as the leaves of the tree; map protocol 2 pieces serve as the nodes of the tree.

When producing an executable file or shared object file, the link editor reformats the contributions to the `.tdesc` section into a single piece with map protocol 1. This tdesc information piece resides in a segment with read permission but without write permission.

When producing an executable file that does not participate in dynamic linking, the link editor defines the symbol `_tdesc` as the address of the tdesc information piece with map protocol 1.

When producing an object file that participates in dynamic linking, the link editor includes a dynamic linking array entry with `d_tag` member equal to `DT_88K_TDESC` and `d_ptr` member equal to the address of the object file's tdesc information piece with map protocol 1. Additionally, in an executable file that participates in dynamic linking, the link editor allocates a tdesc information piece with map protocol 2 and defines the symbol `_tdesc` as the address of this second piece. This second piece resides in a segment with both read and write permissions and has at least as many tdesc piece entries as two more than the number of shared object files referenced by the executable file. Both words of each of the entries are initially zero. The dynamic linker fills in an entry for each object file whose dynamic linking array it processes.

Debug Info

When producing an executable file, the link editor creates the following data structure in a segment with read permission but without write permission and defines the symbol `_debug_info` as the address of the beginning of the structure. (See "Program Header" in Chapter 5 for the segment description.) The structure shall be word-aligned.

Figure 3-48: `_debug_info` **Structure**

Word Position	Bit Range	Interpretation
0	31 - 0	debug info protocol, the integer 1
1	31 - 0	the value of the `_tdesc` symbol
2	31 - 0	number of text words
3	31 - 0	pointer to text words
4	31 - 0	number of data words
5	31 - 0	pointer to data words

The *pointer to text words* is the address of a contiguous sequence of words that reside in a segment with execute permission and that are not otherwise referenced. The *number of text words* indicates the number of such words. The number of text words shall be at least 1. These words are available to a debugger, for use as places to set breakpoints safely for its own use.

The *pointer to data words* is the address of a contiguous sequence of words that reside in a segment with write permission and that are not otherwise referenced. The *number of data words* indicates the number of such words. The number of data words shall be at least 256. These words are available to a debugger, for use as places to store data safely for its own use in the memory of the process being debugged.

When producing an executable file, the link editor shall create a single segment of type `PT_88K_DEBINFADDR`. This segment shall contain a single word, whose value is the value of the `_debug_info` symbol.

4. OBJECT FILES

4. OBJECT FILES

4 OBJECT FILES

ELF Header

Machine Information

For file identification in `e_ident`, the M88000 requires the following values.

Figure 4-1: M88000 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. The M88000 defines no flags, so this member contains zero. Processor identification resides in the ELF header's `e_machine` member and must have the value 5, defined as the name `EM_88K`.

Sections

The M88000 architecture is such that an individual section cannot permit writing and execution attributes—SHF_WRITE and SHF_EXECINSTR—at the same time.

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
`.tdesc`	SHT_PROGBITS	*see below*

`.tdesc`	This section holds text description information. It has the SHF_ALLOC attribute, it does not have the SHF_WRITE attribute, and it may or may not have the SHF_EXECINSTR attribute. See ''Text Description Information'' in Chapter 3 for more information.

Symbol Table

Symbol Values

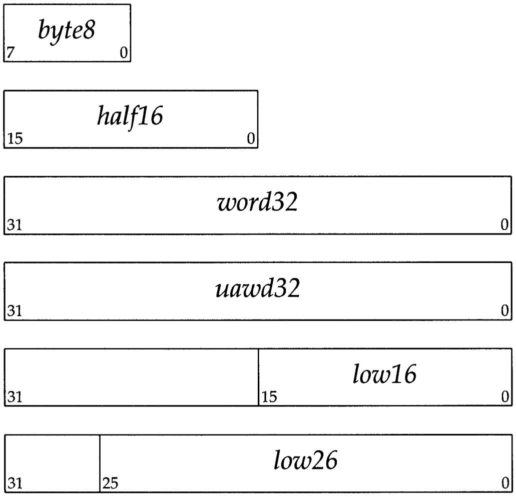
If an executable file contains a reference to a function defined in one of its associated shared objects, the symbol table section for that file will contain an entry for that symbol. The `st_shndx` member of that symbol table entry contains `SHN_UNDEF`. This signals to the dynamic linker that the symbol 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

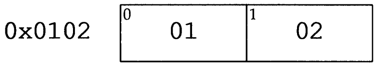
Relocation Types

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

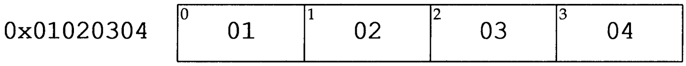


byte8	This specifies an 8-bit field occupying 1 byte with arbitrary alignment.
half16	This specifies a 16-bit field occupying 2 bytes with 2-byte alignment.



word32 This specifies a 32-bit field occupying 4 bytes with 4-byte alignment. These values use the byte order illustrated below.

uawd32 This specifies a 32-bit field occupying 4 bytes with arbitrary alignment. These values use the same byte order as for *word32*.



low16 This specifies a 16-bit field occupying the least significant bits of a field similar to *word32*. These bits represent values in the same byte order as *word32*.

low26 This specifies a 26-bit field occupying the least significant bits of a field similar to *word32*. These bits represent values in the same byte order as *word32*.

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.

AB This means the addressing base for the shared object. See Chapter 5 for more information.

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

	information about the base address.
`G`	This means the place (section offset or address) of a global offset table entry for the symbol. See Chapter 5 for more information.
`GP`	This means the place (section offset or address) of a global offset table procedure entry for the symbol. See Chapter 5 for more information.
`L`	This means the place (section offset or address) of the symbol, or of a procedure linkage table entry for the symbol. See Chapter 5 for more information.
`P`	This means the place (section offset or address) of the storage unit being relocated (computed using `r_offset`).
`S`	This means the value of the symbol whose index resides in the relocation entry.

Relocation entries apply to bytes (*byte8*), halfwords (*half16*), or words (the others). In any case, the `r_offset` value designates the offset or virtual address of the first byte of the affected storage unit. The relocation type specifies which bits to change and how to calculate their values. The M88000 uses only `Elf32_Rela` relocation entries, with explicit addends. Thus the `r_addend` member serves as the relocation addend.

The following general rules apply to the interpretation of the relocation types in Figure 4-4.

- "+" and "–" denote 32-bit modulus addition and subtraction, respectively. ">>" denotes arithmetic right shifting of the value of the left operand by the number of bits given by the right operand.
- For relocation types whose names end in "`_DISP16`", the upper 15 bits of the value computed before shifting must all be the same. For relocation types whose names end in "`_DISP26`", the upper 5 bits of the value computed before shifting must all be the same. For relocation types whose names end in either "`_DISP16`" or "`_DISP26`", the low 2 bits of the value computed before shifting must all be zero.
- For relocation types whose names end in "`_8`", the upper 24 bits of the computed value must all be zero. For relocation types whose names end in "`_8S`", the upper 25 bits of the computed value must all be the same. For relocation types whose names end in "`_16`", the upper 16 bits of the computed value must all be zero. For relocation types whose names end in

"`_16S`", the upper 17 bits of the computed value must all be the same.

- `#hi16(`*value*`)` and `#lo16(`*value*`)` denote the high and low 16 bits, respectively, of the indicated value.
- Reference in a calculation to the value "`G`" implicitly creates a global offset table entry for the indicated symbol. Reference in a calculation to the value "`GP`" implicitly creates a global offset table procedure entry for the indicated symbol. Reference in a calculation to the value "`L`" may implicitly create a procedure linkage table entry for the indicated symbol.
- A relocation type whose calculation involves either the value "`B`" or the value "`AB`" may only be used in a shared object.
- For relocation types whose names begin with either "`R_88K_ABDIFF_`" or "`R_88K_ABREL_`", the symbol's value must represent an address in the shared object containing the relocation.
- For relocation types whose names include "`_SREL_`", the address of the storage unit affected by the relocation either must both be in the same shared object, or must both be in an executable file.
- Where a relocation type does not use the associated symbol, the symbol index in the relocation entry must be zero.
- The link editor shall detect and report violations of restrictions described above.

Figure 4-4: Relocation Types, Part 1 of 2

Name	Value	Field	Calculation
R_88K_NONE	0	none	none
R_88K_COPY	1	none	see below
R_88K_GOTP_ENT	2	*word32*	see below
R_88K_8	4	*byte8*	S + A
R_88K_8S	5	*byte8*	S + A
R_88K_16S	7	*half16*	S + A
R_88K_DISP16	8	*half16*	(S + A - P) >> 2
R_88K_DISP26	10	*low26*	(S + A - P) >> 2
R_88K_PLT_DISP26	14	*low26*	(L + A - P) >> 2
R_88K_BBASED_32	16	*word32*	B + A
R_88K_BBASED_32UA	17	*uawd32*	B + A
R_88K_BBASED_16H	18	*half16*	#hi16(B + A)
R_88K_BBASED_16L	19	*half16*	#lo16(B + A)
R_88K_ABDIFF_32	24	*word32*	AB - S + A
R_88K_ABDIFF_32UA	25	*uawd32*	AB - S + A
R_88K_ABDIFF_16H	26	*half16*	#hi16(AB - S + A)
R_88K_ABDIFF_16L	27	*half16*	#lo16(AB - S + A)
R_88K_ABDIFF_16	28	*half16*	AB - S + A
R_88K_32	32	*word32*	S + A
R_88K_32UA	33	*uawd32*	S + A
R_88K_16H	34	*half16*	#hi16(S + A)
R_88K_16L	35	*half16*	#lo16(S + A)
R_88K_16	36	*half16*	S + A
R_88K_GOT_32	40	*word32*	G + A
R_88K_GOT_32UA	41	*uawd32*	G + A
R_88K_GOT_16H	42	*half16*	#hi16(G + A)
R_88K_GOT_16L	43	*half16*	#lo16(G + A)
R_88K_GOT_16	44	*half16*	G + A
R_88K_GOTP_32	48	*word32*	GP + A
R_88K_GOTP_32UA	49	*uawd32*	GP + A
R_88K_GOTP_16H	50	*half16*	#hi16(GP + A)
R_88K_GOTP_16L	51	*half16*	#lo16(GP + A)
R_88K_GOTP_16	52	*half16*	GP + A

Figure 4-5: Relocation Types, Part 2 of 2

Name	Value	Field	Calculation
R_88K_PLT_32	56	*word32*	L + A
R_88K_PLT_32UA	57	*uawd32*	L + A
R_88K_PLT_16H	58	*half16*	#hi16(L + A)
R_88K_PLT_16L	59	*half16*	#lo16(L + A)
R_88K_PLT_16	60	*half16*	L + A
R_88K_ABREL_32	64	*word32*	S + A - AB
R_88K_ABREL_32UA	65	*uawd32*	S + A - AB
R_88K_ABREL_16H	66	*half16*	#hi16(S + A - AB)
R_88K_ABREL_16L	67	*half16*	#lo16(S + A - AB)
R_88K_ABREL_16	68	*half16*	S + A - AB
R_88K_GOT_ABREL_32	72	*word32*	G + A - AB
R_88K_GOT_ABREL_32UA	73	*uawd32*	G + A - AB
R_88K_GOT_ABREL_16H	74	*half16*	#hi16(G + A - AB)
R_88K_GOT_ABREL_16L	75	*half16*	#lo16(G + A - AB)
R_88K_GOT_ABREL_16	76	*half16*	G + A - AB
R_88K_GOTP_ABREL_32	80	*word32*	GP + A - AB
R_88K_GOTP_ABREL_32UA	81	*uawd32*	GP + A - AB
R_88K_GOTP_ABREL_16H	82	*half16*	#hi16(GP + A - AB)
R_88K_GOTP_ABREL_16L	83	*half16*	#lo16(GP + A - AB)
R_88K_GOTP_ABREL_16	84	*half16*	GP + A - AB
R_88K_PLT_ABREL_32	88	*word32*	L + A - AB
R_88K_PLT_ABREL_32UA	89	*uawd32*	L + A - AB
R_88K_PLT_ABREL_16H	90	*half16*	#hi16(L + A - AB)
R_88K_PLT_ABREL_16L	91	*half16*	#lo16(L + A - AB)
R_88K_PLT_ABREL_16	92	*half16*	L + A - AB
R_88K_SREL_32	96	*word32*	S + A - P
R_88K_SREL_32UA	97	*uawd32*	S + A - P
R_88K_SREL_16H	98	*half16*	#hi16(S + A - P)
R_88K_SREL_16L	99	*half16*	#lo16(S + A - P)

Relocation types with special semantics are described below.

`R_88K_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_88K_GOTP_ENT`	This relocation type assists dynamic linking. The relocation offset gives the location of a global offset table procedure entry. The relocation symbol names the procedure. The relocation addend gives the address of the associated GOTP binding entry. For an executable file, this address is absolute; for a shared object file, it is relative to the base address for the shared object. See Chapter 5 for details.

The use of relocation types whose names end in "`_16`" is generally subject to failure, because the value computed may not fit in 16 bits. However, the use of the `R_88K_GOT_ABREL_16` and `R_88K_GOTP_ABREL_16` relocation types shall not fail unless the total number of distinct GOT and GOTP entries for the executable or shared object being link edited exceeds 16 380. In other words, the link editor is obliged to favor GOT and GOTP entries when choosing an addressing base and laying out the private data of either the executable or shared object file.

5. PROGRAM LOADING AND DYNAMIC LINKING

5. PROGRAM LOADING AND DYNAMIC LINKING

5 PROGRAM LOADING AND DYNAMIC LINKING

Program Header

An additional segment type, `PT_88K_DEBINFADDR`, is defined with value 0x7000 0001. This segment contains a single word whose value is the value of the `_debug_info` symbol. The segment is created by the link editor. It allows a debugger operating as a process separate from the process it is debugging to locate the debug information in the executable file.

Segment Permissions

The M88000 architecture is such that an individual segment cannot permit writing and execution attributes—`PF_W` and `PF_X`—at the same time. The following combinations of segment permissions are valid for the M88000:

Figure 5-1: Segment Permissions

Flags	Value	Permissions Granted		
		Read	Write	Execute
none	0	no	no	no
`PF_X`	1	unspecified	no	yes
`PF_W`	2	unspecified	yes	unspecified
`PF_R`	4	yes	no	unspecified
`PF_R+PF_X`	5	yes	no	yes
`PF_R+PF_W`	6	yes	yes	unspecified

In the table, "yes" indicates the access shall be allowed; "no" indicates the access shall be denied and a `SIGBUS` signal shall be sent to the process; "unspecified" indicates that the process cannot rely on either obtaining access nor receiving the signal.

For the M88000 architecture, the segment permissions indicate only the initial state of the segment. The use of the `mprotect`(KE_OS) function can change the state during execution.

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 M88000 segments are congruent modulo 64K (0x10000). The value of the `p_align` member of each program header in a shared object file must be 64K.

Figure 5-2: Executable File Example

File Offset	File	Virtual Address
0	ELF header	
	Program header table	
	Other information	
`0x100`	Text segment	`0x10100`
	. . .	
	`0x2be00` bytes	`0x3beff`
`0x2bf00`	Data segment	`0x4bf00`
	. . .	
	`0x4e00` bytes	`0x50cff`
`0x30d00`	Other information	
	. . .	

Figure 5-3: Program Header Segments Example

Member	Text	Data
p_type	PT_88K_LOAD	PT_88K_LOAD
p_offset	0x100	0x2bf00
p_vaddr	0x10100	0x4bf00
p_paddr	unspecified	unspecified
p_filesz	0x2be00	0x4e00
p_memsz	0x2be00	0x5e24
p_flags	PF_R+PF_X	PF_R+PF_W
p_align	0x10000	0x10000

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

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

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

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

Figure 5-4: Process Image Segments

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

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

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

Figure 5-5: Example Shared Object Segment Addresses

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

Dynamic Linking

Dynamic Section

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

`DT_PLTGOT`	This entry's `d_ptr` member gives the address of three consecutive words in the private data of an executable or shared object file. These 12 bytes must be 4-byte aligned. The first word must be set by the link editor to contain the address of the symbol `_DYNAMIC`; the address is absolute for an executable file and relative to the base address for a shared object. The second and third words are used to support lazy binding. The `DT_PLTGOT` entry is required in every object file that participates in dynamic linking. The link editor chooses where to locate the three words; one natural place would be the beginning of the global offset table.

`DT_JMPREL`
`DT_PLTRELSZ`
`DT_PLTREL`
	On the M88000, these entries specify a relocation table that pertains to global offset table procedure entries, rather than to the procedure linkage table, as described in the System V ABI. This relocation table should contain all relocation entries of type `R_88K_GOTP_ENT`, and only those entries. In particular, relocation entries applying to the procedure linkage table are found with all other relocation entries in the relocation table specified by the `DT_RELA`, `DT_RELASZ`, and `DT_RELAENT` entries.

The following additional dynamic array tags are defined:

Figure 5-6: Dynamic Array Tags, `d_tag`

Name	Value	`d_un`	Executable	Shared Object
`DT_88K_ADDRBASE`	`0x70000001`	`d_ptr`	ignored	required
`DT_88K_PLTSTART`	`0x70000002`	`d_ptr`	optional	optional
`DT_88K_PLTEND`	`0x70000003`	`d_ptr`	optional	optional
`DT_88K_TDESC`	`0x70000004`	`d_ptr`	optional	optional

`DT_88K_ADDRBASE`

This entry's `d_ptr` member gives the addressing base for the shared object.

`DT_88K_PLTSTART`

This entry's `d_ptr` member gives the low address (inclusive) of the PLT region in an object file.

`DT_88K_PLTEND`

This entry's `d_ptr` member gives the high address (exclusive) of the PLT region in an object file.

`DT_88K_TDESC` This entry's `d_ptr` member gives the address of the tdesc information for the object file. See "Text Description Information" in Chapter 3 for more information.

If either of `DT_88K_PLTSTART` or `DT_88K_PLTEND` is present, both must be present.

The *PLT region* is that portion of an object file that must be made executable by the dynamic linker after relocations are performed in the region. The PLT region includes all PLT entries for the object file that require relocation by the dynamic linker. The region of memory between (((`DT_88K_PLTSTART` value) / 64K) * 64K) (inclusive) and ((((`DT_88K_PLTEND` value) + 64K – 1) / 64K) * 64K) (exclusive), where arithmetic is as for unsigned integers in the C language, is subject to being made executable by the dynamic linker.

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 can reference its global offset table in several ways:

- An executable file can reference its global offset table absolutely, as it would any data, because the address of the global offset table is known to the link editor. A shared object can reference its global offset table with position-independent references, because all of the text and data of a shared object file remains fixed relative to itself no matter where the shared object segments are assigned in memory.
- A shared object typically references its global offset table relative to the shared object's addressing base. The link editor establishes the addressing base and the location of the global offset table, so it can calculate constant offsets to global offset table entries. The addressing base value can be computed by a function in a shared object in a position-independent manner as shown in Figure 3-32.
- References from a shared object's procedure linkage table to the global offset table procedure entries are made absolutely. This is possible because the procedure linkage table is private to the shared object.

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 refer to the global offset table. The dynamic linker determines the associated symbol values, calculates their absolute addresses, and sets the global offset 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.

A global offset table entry provides direct access to the absolute address of a symbol, without compromising position independence and sharability. 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 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.

Global offset table (''GOT'') entries are created by the link editor in response to the use of certain relocation types. A GOT entry is 4 bytes long and 4-byte aligned and is allocated in writable memory private to the executable or shared object file. After relocation by the link editor, the dynamic linker, or both, a GOT entry generally contains the value of its associated symbol, which is usually the address of the entity (object or function) represented by the symbol. The one exception is the case of a function for which there is a PLT entry in the executable file. In this case the GOT entry contains the address of that PLT entry. In this way, the address by which the executable file knows the function (its PLT entry address) is also the address by which all shared objects know the function.

More efficient access to functions is provided by special GOT entries known as ''global offset table procedure'' (''GOTP'') entries. Like GOT entries, GOTP entries are created by the link editor in response to use of certain relocation types, are 4 bytes long and 4-byte aligned, are allocated in writable memory private to the executable or shared object file, and are relocated by the link editor, dynamic linker, or both. A GOTP entry, however, may only refer to a function. During execution, the GOTP entry contains an address to which control can be transferred in order to reach the function represented by the symbol associated with the GOTP entry. Moreover, the contents of the GOTP entry may change during execution. This is ''lazy binding'', described below. Although the contents of a GOTP entry may change during execution, every value contained in a GOTP entry serves to transfer control correctly to the associated function.

A GOTP entry has an associated relocation of type `R_88K_GOTP_ENT`. The relocation information and the initial contents of the entry are described under the `R_88K_GOTP_ENT` relocation type.

There are two separate relocation operations that the dynamic linker may perform for a GOTP entry. The first, called ''pre-binding,'' is performed during the dynamic linker's relocation phase when lazy binding is in effect (when the `LD_BIND_NOW` environment variable is missing or null). In pre-binding, the dynamic linker rewrites the GOTP entry so that calling through it invokes the dynamic linker. When the first invocation is made through the GOTP entry, the dynamic linker gains control and performs the second relocation operation on the GOTP entry, called "binding." Binding involves locating the relocation table entry associated with the GOTP entry, looking up the associated symbol to find where the function resides in memory, rewriting the GOTP entry to point directly to the function, and finally transferring control to the function. If lazy binding is not in effect (the value of the `LD_BIND_NOW` environment variable is non-null), the dynamic linker simply performs the binding operation during its relocation phase, bypassing the pre-binding step altogether.

NOTE Lazy binding generally improves overall application performance, because unused symbols incur lower dynamic linking cost. 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.

The link editor and dynamic linker collaborate to support lazy binding. For each GOTP entry, the link editor creates a "GOTP binding" entry, a sequence of instructions that serves to transfer control to the dynamic linker. When lazy binding is in effect, the dynamic linker stores the address of the GOTP binding entry in the GOTP entry. (The addend in the relocation entry for the GOTP entry locates the GOTP binding entry.) The dynamic linker also stores a word identifying the executable or shared object file and the address of its binding routine in the second and third words, respectively, of the three words located by the `DT_PLTGOT` value for the executable or shared object file.

The GOTP binding entry is responsible for transferring control to the address contained in the word at "`DT_PLTGOT` value" + 8, having extended the stack by 16 bytes with the following values:

Figure 5-7: **GOTP Binding Entry Stack Frame**

`#r31` Offset	Contents
12	`#r1` value at time of call
8	`reloc_off` value
4	word at "`DT_PLTGOT` value" + 4
0	the value 0

The `reloc_off` value is the offset, in bytes, from the `DT_JMPREL` value for the executable or shared object file containing the GOTP entry, to the relocation entry for the GOTP entry.

The GOTP binding entry may destroy the contents of register `#r11`. The GOTP binding entry, in transferring to the dynamic linker, must place an appropriate return address in `#r1`, to maintain a proper return address chain for text description information purposes.

There are many ways for the link editor to satisfy the above requirements. One possible implementation of the GOTP binding entry is:

Figure 5-8: **GOTP Binding Entry**

```
        or.u    #r11,#r0,#hi16(reloc_off)
        or      #r11,#r11,#lo16(reloc_off)
        br      GOTP_binding_helper
```

where `GOTP_binding_helper` is a sequence of instructions particular to the given executable or shared object file. A GOTP binding helper routine that cooperates with GOTP binding entries as shown above could be:

Figure 5-9: GOTP Binding Helper

```
         subu   #r31,#r31,16
         st     #r1,#r31,12
         st     #r11,#r31,8
         bsr    here
here:    or.u   #r11,#r0,#hi16(DT_PLTGOT-here)
         or     #r11,#r11,#lo16(DT_PLTGOT-here)
         addu   #r11,#r11,#r1
         ld     #r1,#r11,4
         st     #r1,#r31,4
         ld     #r11,#r11,8
         st     #r0,#r31,0
         jsr    #r11
         or     #r0,#r0,#r0
```

The expression "`DT_PLTGOT-here`" represents the distance from label `here` to the `DT_PLTGOT`-specified value. The final "no-op" instruction is needed so that the return address placed in `#r1` by the `jsr` instruction will correctly locate the GOTP binding helper routine for text description information purposes.

The example sequences shown for the GOTP binding entry and GOTP binding helper routine are designed not to require any relocation by the dynamic linker. Hence, they can be part of the normal text of a shared object. In particular, they don't need to reside along with PLT entries in the PLT region. However, it may be convenient for the link editor to create a procedure linkage table consisting of the GOTP binding helper routine followed by PLT and GOTP binding entries for each GOTP entry.

Function Addresses

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

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

- 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.
- 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.
- 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

The procedure linkage table is a repository for short sequences of code that provide convenient access to GOTP entries. A procedure linkage table (''PLT'') entry is a sequence of instructions that passes control on to a procedure identified by a particular GOTP entry. The benefit of a PLT entry is that it provides an address (the address of its first instruction) to which control can simply be transferred (as by a `bsr` instruction, for example) in order to invoke a GOTP entry with the appropriate protocol.

It is usually better to access a GOTP entry directly rather than indirectly through a PLT entry. However, there are some situations in which a PLT entry can be useful.

- When code is compiled for inclusion in an executable file (and, in particular, not for inclusion in a shared object), it is generally best to compile a call into simply a `bsr` instruction, under the assumption that most calls from outside of all shared objects will be to procedures that are not in a shared object. If it turns out for such a call that the procedure being called is in a shared object, a PLT entry can be created by the link editor, and the `bsr` instruction can simply be adjusted to reference the PLT entry.
- When code is compiled for inclusion in a shared object, the compiler can emit instructions to access the GOTP entry directly. It may be useful, however, for either convenience of the compiler or compactness of the call (when many are made statically to the same GOTP entry), to use simply a `bsr` instruction and a PLT entry.

The procedure linkage table is unlike a normal table in one respect—its entries are not necessarily all the same size. (Nevertheless, typically the entries will all be the same size.) The form of a typical PLT entry, for a hypothetical procedure named ''name,'' is shown below, as if it were written in assembly language.

Figure 5-10: PLT Entry

```
name:   or.u   #r11,#r0,#hi16(name#gotp)
        ld     #r11,#r11,#lo16(name#gotp)
        jmp    #r11
```

Although the instruction sequence shown above is only one of many possible sequences, the following points will invariably be true:

- The GOTP entry for the procedure is referenced absolutely. Because the global offset table for a shared object may reside at different locations in different processes, the PLT entry code cannot be shared by different processes.
- Register `#r11` is used to load the contents of the GOTP entry.
- No register other than `#r11` is changed by the PLT entry sequence.

Executable files and shared object files have separate procedure linkage tables, just as they have separate global offset tables. The treatment by the link editor and dynamic linker can vary in the two different cases. The procedure linkage table in an executable file can be relocated by the link editor, so it can be placed in the text area and shared by all processes executing that file. Note that, in this case, the dynamic linker doesn't act on the procedure linkage table at all. Because the PLT entry refers to absolute addresses in the global offset table, however, the procedure linkage table in a shared object file cannot be relocated until the shared object has had its memory assigned by the dynamic linker. In the shared object case, the link editor constructs the procedure linkage table in a segment that is initially writable but not executable. The link editor records the extent of the PLT region with the `DT_88K_PLTSTART` and `DT_88K_PLTEND` information. The dynamic linker loads the shared object, performs relocations (including those on the procedure linkage table), then uses `mprotect`(KE_OS) to change the segment containing the procedure linkage table from writable to executable. Note that the area of memory subject to being changed from writable to executable is the area containing the PLT region, rounded outward on each end to a 64K boundary.

The link editor is responsible for contributing text description information to describe the code that it creates, namely the PLT entries, GOTP binding entries, and GOTP binding helper routine.

6. LIBRARIES

6. LIBRARIES

6 LIBRARIES

System Library

Additional Entry Points

There are no additional entry points required by the Motorola 88000 Processor Supplement.

Support Routines

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

Figure 6-1: libsys Support Routines

```
getpsr   sbrk   setpsr
```

`unsigned getpsr(void);`

This function returns the current contents of the Processor Status Register (PSR).

`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.

`unsigned setpsr(unsigned psr);`

This function sets several bits in the Processor Status Register (PSR) of the calling process. These bits control certain aspects of the execution of the process. The bits that can be set are the SER, C, BO, and MXM bits; the precise semantics of these bits are defined in the *MC88100 User's Manual*.

The parameter `psr` is the bitwise inclusive OR of one or more of the following values: `PSR_SER`, `PSR_C`, `PSR_MXM`, or `PSR_BO`. (See `<m88kbcs.h>`.)

Setting the SER bit (`PSR_SER`) turns on serial mode. Clearing this bit allows concurrent operation.

Setting the C (`PSR_C`) bit sets the carry bit to one; clearing this bit zeroes the carry bit.

Setting the MXM bit (`PSR_MXM`) disables misaligned access exceptions. Clearing this bit enables misaligned access exceptions; in this mode a misaligned access causes the system to deliver a `SIGBUS` signal to the process.

Setting the BO bit (`PSR_BO`) causes the current byte order to be Little-Endian; clearing the BO bit causes the current byte order to be Big-Endian. Regardless of the setting of the BO bit, all interfaces to or from the system are always in Big-Endian order: all fields in structures, signal frames, etc.

All bits in the `psr` parameter except SER, C, BO, and MXM are ignored.

The `setpsr` call returns the previous value of the Processor Status Register.

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 88000 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 88000 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 88000 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
```

NOTE The following `struct flock` is defined differently than in the 88open Object Compatibility Standard.

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;
};
```

NOTE The following `struct ipc_perm` is defined differently than in the 88open Object Compatibility Standard.

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: `<sys/m88kbcs.h>`

```
#define PSR_SER       0x20000000
#define PSR_C         0x10000000
#define PSR_MXM       0x00000004
#define PSR_BO        0x40000000
```

Figure 6-22: `<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 6-23: `<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 6-24: `<mon.h>`

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

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

typedef unsigned short WORD;
```

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

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

NOTE The following `struct msqid_ds` is defined differently than in the 88open Object Compatibility Standard.

Figure 6-26: `<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 6-27: `<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 6-28: `<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 6-29: `<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 6-30: `<nl_types.h>`

```
#define NL_SETD        1

typedef short nl_item ;

typedef void *nl_catd;
```

Figure 6-31: `<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 6-32: `<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 6-33: `<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 6-34: `<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 6-35: `<sys/regset.h>`**, Part 1 of 2**

```
typedef unsigned int   greg_t;
#define NGREG          38
typedef greg_t         gregset_t[NGREG];

#define R_R0    0
#define R_R1    1
#define R_R2    2
#define R_R3    3
#define R_R4    4
#define R_R5    5
#define R_R6    6
#define R_R7    7
#define R_R8    8
#define R_R9    9
#define R_R10   10
#define R_R11   11
#define R_R12   12
#define R_R13   13
#define R_R14   14
#define R_R15   15
#define R_R16   16
#define R_R17   17
#define R_R18   18
#define R_R19   19
#define R_R20   20
#define R_R21   21
#define R_R22   22
#define R_R23   23
#define R_R24   24
#define R_R25   25
```

Figure 6-36: `<sys/regset.h>`**, Part 2 of 2**

```
#define R_R26  26
#define R_R27  27
#define R_R28  28
#define R_R29  29
#define R_R30  30
#define R_R31  31
#define R_XIP  32
#define R_NIP  33
#define R_FIP  34
#define R_PSR  35
#define R_FPSR 36
#define R_FPCR 37

typedef struct dfltinfo {
        unsigned int  dma;
        unsigned int  dmt;
        unsigned int  dmd;
} dfltinfo_t;

typedef struct fpifltinfo {
        unsigned int  fprh;
        unsigned int  fprl;
        unsigned int  fpit;
} fpifltinfo_t;
```

Figure 6-37: `<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 6-38: `<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 6-39: `<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 6-40: `<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 6-41: `<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 6-42: `<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 6-43: `<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 6-44: `<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 6-45: `<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 6-46: `<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 6-47: `<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 6-48: `<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 6-49: `<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 6-50: `<search.h>`

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

NOTE The following `struct semid_ds` is defined differently than in the 88open Object Compatibility Standard.

Figure 6-51: `<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 6-52: `<setjmp.h>`

```
#define _JBLEN 40
#define _SIGJBLEN 128
typedef int jmp_buf[_JBLEN];
typedef int sigjmp_buf[_SIGJBLEN];
```

NOTE The following `struct shmid_ds` is defined differently than in the 88open Object Compatibility Standard.

Figure 6-53: `<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 6-54: `<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 6-55: `<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_PRVOPC      2

#define FPE_INTOVF      0x80000001
#define FPE_INTDIV      0x80000002
#define FPE_FLTSUB      0x80000003
#define FPE_FLTRES      0x01
#define FPE_FLTOVF      0x02
#define FPE_FLTUND      0x04
#define FPE_FLTDIV      0x08
#define FPE_FLTINV      0x10
#define FPE_PRIVVIO     0x20
#define FPE_UNIMPL      0x40
#define FPE_FLTNAN      0x80

#define SEGV_MAPERR     0x01
#define SEGV_ACCERR     0x02
#define SEGV_CODE       0x04
#define SEGV_DATA       0x08

#define BUS_ADRALN      0x01
#define BUS_ADRERR      0x02
#define BUS_OBJERR      0x03
#define BUS_ALIGN       0x04
#define BUS_PROT        0x08
```

Figure 6-56: `<sys/siginfo.h>`**, Part 2 of 3**

```
#define CLD_EXITED     1
#define CLD_KILLED     2
#define CLD_DUMPED     3
#define CLD_TRAPPED    4
#define CLD_STOPPED    5
#define CLD_CONTINUED  6

#define POLL_IN        1
#define POLL_OUT       2
#define POLL_MSG       3
#define POLL_ERR       4
#define POLL_PRI       5
#define POLL_HUP       6

#define SI_MAXSZ       256
#define SI_PAD         ((SI_MAXSZ/sizeof(int))-4)

typedef struct {
        int     eb_signo;
        int     eb_code;
        union {
                int     _pad[14];
                dfltinfo_t     _fault;
                fpifltinfo_t   _fpui;
        } _eb_registers;
} exblk_t;
```

Figure 6-57: `<sys/siginfo.h>`**, Part 3 of 3**

```
typedef struct siginfo {
        int     si_signo;
        int     si_errno;
        int     si_code;
        int     si_machinexcep;
        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 {
                        int     _fd;
                        long    _band;
                } _file;
                struct {
                        int     _ncodes;
                        exblk_t*_exblks;
                } _machine;
        } _data;
} siginfo_t;
```

Figure 6-58: `<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 6-59: `<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 6-60: `<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 6-61: `<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 6-62: `<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 6-63: `<stdarg.h>`

```
typedef struct {
        int             next_arg;
        int             *mem_ptr;
        int             *reg_ptr;
        } va_list;
```

The member `next_arg` is the number of words from the beginning of the argument list to the beginning of the next argument to be returned by `va_arg`. `next_arg` shall always have a nonnegative value. `mem_ptr` points at the beginning of the argument area. `reg_ptr` points at a structure of the following form:

```
struct {int #r2, #r3, #r4, #r5, #r6, #r7, #r8, #r9;}
```

where each member contains the value at procedure entry of the indicated register, if that register holds a portion of the variable argument list represented by the `va_list` structure. A procedure receiving a `va_list` structure shall not refer to members of the structure pointed at by `reg_ptr` that do not correspond to portions of the variable argument list that the `va_list` structure represents. The structure pointed at by `reg_ptr` shall be 8-byte aligned.

NOTE The procedure using the `va_list` structure determines, for each argument of the variable argument list, whether to fetch the argument value from the memory area or the register area, according to the position of the argument in the argument list and the type of the argument (including size, alignment, and whether it is a structure or union).

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

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

Figure 6-65: `<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 6-66: `<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 6-67: `<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 6-68: `<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 6-69: `<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 6-70: `<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 6-71: `<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 6-72: `<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 6-73: `<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 6-74: `<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 6-75: `<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 6-76: `<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 6-77: `<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 6-78: `<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 6-79: `<sys/times.h>`

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

Figure 6-80: `<sys/tiuser.h>`, **Service Types**

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

Figure 6-81: `<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 6-82: `<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 6-83: `<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 6-84: `<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 6-85: `<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 6-86: `<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 6-87: `<sys/tiuser.h>`, **Fields of Structures**

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

Figure 6-88: `<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 6-89: `<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 6-90: `<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 6-91: `<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[210];
} ucontext_t;

#define GETCONTEXT      0
#define SETCONTEXT      1
```

Figure 6-92: `<uio.h>`

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

Figure 6-93: `<ulimit.h>`

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

Figure 6-94: `<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 6-95: `<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_BCS_VERSION      9
#define _SC_BCS_VENDOR_STAMP 10
#define _SC_BCS_SYS_ID       11
#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 6-96: `<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 6-97: `<utime.h>`

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

Figure 6-98: `<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 6-99: `<varargs.h>`

```
#include <stdarg.h>
```

Figure 6-100: `<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

320–167

UNIX
PRESS

A Prentice Hall Title

ISBN 0-13-877655-5