SYSTEM V

APPLICATION BINARY INTERFACE

Generic 64-Bit Extensions

August 1, 1994

Delta Document 1.33

DRAFT

SPARC International Confidential

Revision History

Rev	Date	Author	Description
-0	10/01/91	HaL ABI Working Group	First Draft
-0.0	10/15/91	HaL ABI Working Group	Added Preface
-1.00	01/02/92	SI ABI Working Group	First Review
-1.10	02/02/92	SI ABI Working Group	Merged 32 and 64-bit ELF/Minor Editing Changes
-1.20	03/07/92	SI ABI Working Group	Updated Chapter 6.
1.30	N/A	Dock Williams, Sun	N/A
1.31	1994-07-13	Howard Gayle, HaL	Changed section 5.4.4 (Shared Object Dependencies) to search
			for shared objects in a subdirectory whose name is derived from
			the machine name.
1.32	1994-07-15	Howard Gayle, HaL	Rewrote section 5.4.4 (Shared Object Dependencies) to add
			rationale and provide three different options.
1.33	1994-08-01	Howard Gayle, HaL	Put section 5.4.4 in brackets and added a paragraph asking the
			ABICC for guidance.

Revision History

Rev	Date	Author	Description

0. PREFACE

0.1. Introduction

The purpose of this document is to describe the differences between the 32-bit generic ABI, as published by AT&T as *System V Application Binary Interface*, and the proposed 64-bit version of the generic ABI.

0.2. Basic Assumptions

A number of basic assumptions are reflected in the proposals presented. It is assumed that it is important to permit the simultaneous support of both 32 and 64-bit binaries but it is not necessary to specifically require 32-bit compatibility. This means it should be possible for 64-bit systems to support both the 64-bit ABI and the 32-bit ABI or just the 64-bit ABI. It is also assumed that most networking software will continue to use 32-bit protocols.

1. INTRODUCTION

1.1. System V Application Binary Interface

[The names of documents in this section may have to change.]

1.2. Foundations and Structure of the ABI

[The names of documents in this section may have to change.]

1.3. How to Use the System V ABI

[The names of documents in this section may have to change.]

1.4. Definitions of Terms

[This section is unchanged.]

2. SOFTWARE INSTALLATION

2.1. Software Installation and Packaging

[We may need to add a new 64-bit tape archive definition.]

2.2. File Formats

[This section is unchanged.]

2.3. File Tree for Add-on Software

[This section is unchanged.]

2.4. Commands that Install, Remove and Access Packages

[This section is unchanged.]

3. LOW_LEVEL SYSTEM INFORMATION

3.1. Introduction

[This section is unchanged.]

3.2. Character Representations

[The following paragraph should be removed from this section.]

- ~~Multibyte character encodings with values above 127 should contain only bytes with values outside the range of 0 to 127. That is, a character set that uses more than one byte per character should not "embed" a byte resembling a 7-bit ASCII character within a multi-byte, non-ASCII character.~~

3.3. Machine Interface (Processor-Specific)

[This section is unchanged.]

3.4. Function Calling Sequence (Processor-Specific)

[This section is unchanged.]

3.5. Operating System Interface (Processor-Specific)

[This section is unchanged.]

3.6. Coding Examples (Processor-Specific)

[This section is unchanged.]

4. OBJECT FILES

4.1. Introduction

4.1.1. File Format

[This section is unchanged.]

4.1.2. Data Representation

As described here, the object file *format* supports various processors with 8-bit bytes and 32 or 64-bit architectures. Nevertheless, it is intended to be extensible to larger (or smaller) architectures. Object files therefore represent some control data with a machine-independent format, making it possible to identify object files and interpret their contents in a common way. Remaining data in an object file uses the encoding of the target processor, regardless of the machine on which the file was created.

Figure 4-2: 32-Bit Data Types

Name	Size	Alignment	Purpose
`Elf32_Addr`	4	4	Unsigned program address
`Elf32_Half`	2	2	Unsigned medium integer
`Elf32_Off`	4	4	Unsigned file offset
`Elf32_Sword`	4	4	Signed large integer
`Elf32_Word`	4	4	Unsigned large integer
`unsigned char`	1	1	Unsigned small integer

Figure 4-2+: 64-Bit Data Types

Name	Size	Alignment	Purpose
`Elf64_Addr`	8	8	Unsigned program address
`Elf64_Half`	2	2	Unsigned small integer
`Elf64_Off`	8	8	Unsigned file offset
`Elf64_Sword`	4	4	Signed medium integer
`Elf64_Sxword`	8	8	Signed large integer
`Elf64_Word`	4	4	Unsigned medium integer
`Elf64_Xword`	8	8	Unsigned large integer

All data structures that the object file format defines follow the “natural” size and alignment guidelines for the relevant class. If necessary, data structures contain explicit padding to ensure 8-byte alignment for 8-byte objects, to force structure sizes to a multiple of 8, etc. Data also have suitable alignment from the beginning of the file. Thus, for example, a structure containing an `Elf64_Addr` member will be aligned on a 8-byte boundary within the file.

For portability reasons, ELF uses no bit-fields.

4.2. ELF Headers

Some object file control structures can grow, because the ELF header contains their actual sizes. If the object file format changes, a program may encounter control structures that are larger or smaller than expected. Programs might therefore ignore “extra” information. The treatment of “missing” information depends on context and will be specified when and if extensions are defined.

Figure 4-3: 32-bit ELF Header (`ELFCLASS32`)

```
typedef struct {
          unsigned char       e_ident[EI_NIDENT];
          Elf32_Half          e_type;
          Elf32_Half          e_machine;
          Elf32_Word          e_version;
          Elf32_Addr          e_entry;
          Elf32_Off           e_phoff;
          Elf32_Off           e_shoff;
          Elf32_Word          e_flags;
          Elf32_Half          e_ehsize;
          Elf32_Half          e_phentsize;
          Elf32_Half          e_phnum;
          Elf32_Half          e_shentsize;
          Elf32_Half          e_shnum;
          Elf32_Half          e_shstrndx;
} Elf32_Ehdr;
```

Figure 4-3+: 64-bit ELF Header (`ELFCLASS64`)

```
typedef struct {
          unsigned char       e_ident[EI_NIDENT];
          Elf64_Half          e_type;
          Elf64_Half          e_machine;
          Elf64_Word          e_version;
          Elf64_Addr          e_entry;
          Elf64_Off           e_phoff;
          Elf64_Off           e_shoff;
          Elf64_Word          e_flags;
          Elf64_Half          e_ehsize;
          Elf64_Half          e_phentsize;
          Elf64_Half          e_phnum;
          Elf64_Half          e_shentsize;
          Elf64_Half          e_shnum;
          Elf64_Half          e_shstrndx;
} Elf64_Ehdr;
```

4.2.1. ELF Identification

As mentioned above, ELF provides an object file framework to support multiple processors, multiple data encodings, and multiple classes of machines. To support this object file family, the initial bytes of the file specify how to interpret the file, independent of the processor on which the inquiry is made and independent of the file's remaining contents.

The initial bytes of an ELF header (and an object file) correspond to the `e_ident` member.

Figure 4-4: `e_ident[]` Identification Indexes

Name	Value	Purpose
`EI_MAG0`	0	File identification
`EI_MAG1`	1	File identification
`EI_MAG2`	2	File identification
`EI_MAG3`	3	File identification
`EI_CLASS`	4	File class
`EI_DATA`	5	Data encoding
`EI_VERSION`	6	File version
`EI_PAD`	7	Start of padding bytes
`EI_NIDENT`	16	Size of `e_ident[]`

These indexes access bytes that hold the following values.

`EI_MAG0` to `EI_MAG3`
A files first 4 bytes hold a "magic number", identifying the file as an ELF object file.

Name	Value	Position
`ELFMAG0`	`0x7f`	`e_ident[EI_MAG0]`
`ELFMAG1`	`'E'`	`e_ident[EI_MAG1]`
`ELFMAG2`	`'L'`	`e_ident[EI_MAG2]`
`ELFMAG3`	`'F'`	`e_ident[EI_MAG3]`

`EI_CLASS` The next byte, `e_ident[EI_CLASS]`, identifies the file's class, or capacity.

Name	Value	Meaning
`ELFCLASSNONE`	0	Invalid Class
`ELFCLASS32`	1	32-bit objects
`ELFCLASS64`	2	64-bit objects

The file format is designed to be portable among machines of various sizes, without imposing the sizes of the largest machine on the smallest. Class `ELFCLASS32` supports machines with files and address spaces up to 4 gigabytes. Class `ELFCLASS64` supports machines with files and virtual address spaces up to 16 exabytes.

Other classes will be defined as necessary, with different basic types and sizes for object file data.

`EI_DATA` Byte `e_ident[EI_DATA]` specifies the data encoding of the processor-specific data in the object file. The following encodings are currently defined.

Name	Value	Meaning
`ELFDATANONE`	0	Invalid data encoding
`ELFDATA2LSB`	1	See below
`ELFDATA2MSB`	2	See below

More information on these encodings appears below. Other values are reserved and will be assigned to new encodings as necessary.

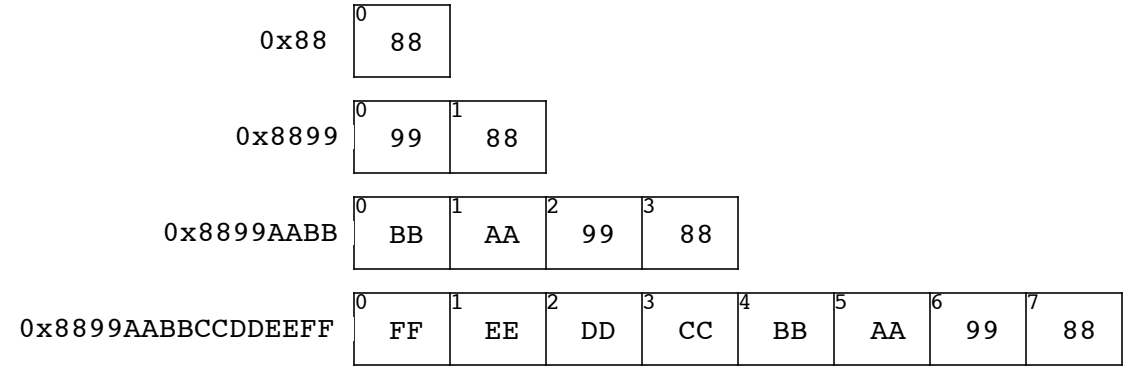
`EI_VERSION` Byte `e_ident[EI_VERSION]` specifies the ELF header version number. Currently this value must be `EV_CURRENT`, as explained for `e_version`.

`EI_PAD` This value marks the beginning of the unused bytes in `e_ident`. These bytes are reserved and set to zero; programs that read object files should ignore them. The value `EI_PAD` will change in the future if currently unused bytes are given meanings.

A file's data encoding specifies how to interpret the basic objects in a file. As described above, class ELF files use objects that occupy 1, 2, 4 and 8 bytes. Under defined encodings, objects are represented as shown below. Byte numbers appear in the upper left corners.

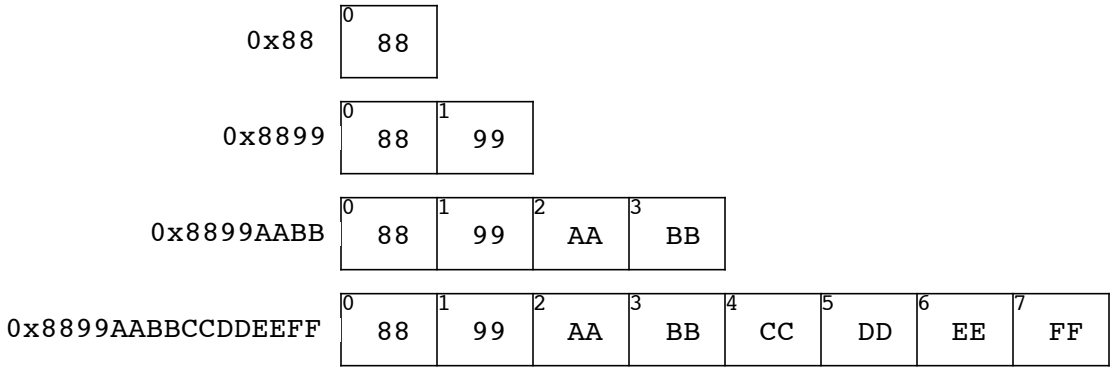
Encoded `ELFDATA2LSB` specifies 2's complement values, with least significant byte occupying the lowest address.

Figure 4-5: Data Encoding `ELFDATA2LSB`



Encoding `ELFDATA2MSB` specifies 2's complement values with the most significant byte occupying the lowest address.

Figure 4-6: Data Encoding `ELFDATA2MSB`



4.2.2. Machine Information (Processor-Specific)

[This section is unchanged.]

4.3. Sections

[Change first sentence to appear as follows:]

An object file's section header table lets one locate all the file's sections. The section header table is an array of `Elf32_Shdr` or `Elf64_Shdr` structures as described below. A section header table index is a subscript into this array. The ELF header's `e_shoff` member gives the byte offset from the beginning of the file to the section header table; `e_shnum` tells how many entries the section header table contains; `e_shentsize` gives the size in bytes of each entry.

Figure 4-8: Section Header (`ELFCLASS32`)

```
typedef struct {
          Elf32_Word          sh_name;
          Elf32_Word          sh_type;
          Elf32_Word          sh_flags;
          Elf32_Addr          sh_addr;
          Elf32_Off           sh_offset;
          Elf32_Word          sh_size;
          Elf32_Word          sh_link;
          Elf32_Word          sh_info;
          Elf32_Word          sh_addralign;
          Elf32_Word          sh_entsize;
} Elf32_Shdr;
```

Figure 4-8+: Section Header (`ELFCLASS64`)

```
typedef struct {
          Elf64_Word          sh_name;
          Elf64_Word          sh_type;
          Elf64_Xword         sh_flags;
          Elf64_Addr          sh_addr;
          Elf64_Off           sh_offset;
          Elf64_Xword         sh_size;
          Elf64_Word          sh_link;
          Elf64_Word          sh_info;
          Elf64_Xword         sh_addralign;
          Elf64_Xword         sh_entsize;
} Elf64_Shdr;
```

4.4. Symbol table

Figure 4-15: Symbol Table Entry (`ELFCLASS32`)

```
typedef struct {
          Elf32_Word          st_name;
          Elf32_Addr          st_value;
          Elf32_Word          st_size;
          unsigned char       st_info;
          unsigned char       st_other;
          Elf32_Half          st_shndx;
} Elf32_Sym;
```

.

Figure 4-15+: Symbol Table Entry (`ELFCLASS64`)

```
typedef struct {
          Elf64_Word          st_name;
          unsigned char       st_info;
          unsigned char       st_other;
          Elf64_Half          st_shndx;
          Elf64_Addr          st_value;
          Elf64_Xword         st_size;
} Elf64_Sym;
```

4.4.1. Relocation Entries

Relocation is the process of connecting symbolic references with symbolic definitions. For example, when a program calls a function, the associated call instruction must transfer control to the proper destination address at execution. In other words, relocatable files must have information that describes how to modify their section contents, thus allowing executable and shared object files to hold the right information for a process's program image. *Relocation entries* are these data.

Figure 4-19: Relocation Entries (`ELFCLASS32`)

```
typedef struct {
          Elf32_Addr          r_offset;
          Elf32_Word          r_info;
} Elf32_Rel;

typedef struct {
          Elf32_Addr          r_offset;
          Elf32_Word          r_info;
          Elf32_Sword         r_addend;
} Elf32_Rela;
```

Figure 4-19+: Relocation Entries (`ELFCLASS64`)

```
typedef struct {
          Elf64_Addr          r_offset;
          Elf64_Xword         r_info;
} Elf64_Rel;

typedef struct {
          Elf64_Addr          r_offset;
          Elf64_Xword         r_info;
          Elf64_Sxword        r_addend;
} Elf64_Rela;
```

`r_offset` This member gives the location at which to apply the relocation action. For a relocatable file, the value is the byte offset from the beginning of the section to the storage unit affected by the relocation. For an executable file or a shared object, the value is the virtual address of the storage unit affected by the relocation.

`r_info` This member gives both the symbol table index with respect to which the relocation must be made, and the type of relocation to apply. For example, a call instruction's relocation entry would hold the symbol table index of the function being called. If the index is `STN_UNDEF`, the undefined symbol index, the relocation uses 0 as the "symbol value". Relocation types are processor-specific; descriptions of their behavior appear in the processor supplement. When the text in the processor supplement refers to a relocation entry's relocation type it means the result of applying `ELF32_R_TYPE` or `ELF64_R_TYPE` to

the entry's `r_info` member. When the text refers to a relocation entry's symbol table index it means the result of applying `ELF32_R_SYM` or `ELF64_R_SYM` to the entry's `r_info` member.

```
#define ELF32_R_SYM(info)           ((info)>>8)
#define ELF32_R_TYPE(info)          ((unsigned char)(info))
#define ELF32_R_INFO(sym, type)     (((sym)<<8)
                                    + (unsigned char)(type))

#define ELF64_R_SYM(info)           ((info)>>32)
#define ELF64_R_TYPE(info)          ((Elf64_Word)(info))
#define ELF64_R_INFO(sym, type)     (((Elf64_Xword)(sym)<<32)
                                    + (Elf64_Xword)(type))
```

`r_addend` This member specifies a constant addend used to compute the value to be stored into the relocatable field.

As shown above, only `ELF32_Rela` and `ELF64_Rela` entries contain an explicit addend. Entries of type `Elf32_Rel` and `Elf64_Rel` store an implicit addend in the location to be modified. Depending on the processor architecture, one form or the other might be necessary or more convenient. Consequently, an implementation for a particular machine may use one form exclusively or either form depending on context.

A relocation section references two other sections: a symbol table and a section to modify. The section header's `sh_info` and `sh_link` members, described in "Sections" above, specify these relationships. Relocation entries for different object files have slightly different interpretations of the `r_offset` member.

- In relocatable files, `r_offset` holds a section offset. That is, the relocation section itself describes how to modify another section in the file; relocation offsets designate a storage unit within the second section.
- In executable and shared object files, `r_offset` holds a virtual address. To make these files' relocation entries more useful for the dynamic linker, the section offset (file interpretation) gives way to a virtual address (memory interpretation).

Although the interpretation of `r_offset` changes for different object files to allow efficient access by the relevant programs, the relocation types' meanings stay the same.

4.4.2. Relocation Types (Processor-Specific)

[This section is unchanged.]

5. PROGRAM LOADING AND DYNAMIC LINKING

5.1. Introduction

[This section is unchanged.]

5.2. Program Header

Figure 5-1: Program Header (`ELFCLASS32`)

```
typedef struct {
          Elf32_Word          p_type;
          Elf32_Off           p_offset;
          Elf32_Addr          p_vaddr;
          Elf32_Addr          p_paddr;
          Elf32_Word          p_filesz;
          Elf32_Word          p_memsz;
          Elf32_Word          p_flags;
          Elf32_Word          p_align;
} Elf32_Phdr;
```

Figure 5-1+: Program Header (`ELFCLASS64`)

```
typedef struct {
          Elf64_Word          p_type;
          Elf64_Word          p_flags;
          Elf64_Off           p_offset;
          Elf64_Addr          p_vaddr;
          Elf64_Addr          p_paddr;
          Elf64_Xword         p_filesz;
          Elf64_Xword         p_memsz;
          Elf64_Xword         p_align;
} Elf64_Phdr;
```

5.2.1. Base Address

[This section is unchanged.]

5.2.2. Segment Permissions

[This section is unchanged.]

5.2.3. Segment Contents

[This section is unchanged.]

5.2.4. Note Section

[This section is unchanged.]

5.3. Program Loading (Processor-Specific)

[This section is unchanged.]

5.4. Dynamic Linking

[This section is unchanged.]

5.4.1. Program Interpreter

[This section is unchanged.]

5.4.2. Dynamic Linker

[This section is unchanged.]

5.4.3. Dynamic Section

Figure 5-9: Dynamic Structure (`ELFCLASS32`)

```
typedef struct {
          Elf32_Sword         d_tag;
          union {
                    Elf32_Word      d_val;
                    Elf32_Addr      d_ptr;
          } d_un;
} Elf32_Dyn;

extern Elf32_Dyn _DYNAMIC[];
```

Figure 5-9+: Dynamic Structure (`ELFCLASS64`)

```
typedef struct {
          Elf64_Xword         d_tag;
          union {
                    Elf64_Xword      d_val;
                    Elf64_Addr       d_ptr;
          } d_un;
} Elf64_Dyn;

extern Elf64_Dyn _DYNAMIC[];
```

5.4.4. Shared Object Dependencies

[For commercial reasons, it will sometimes be desirable for one vendor-supplied system to support two or more ABI interfaces. For example, all vendors of SPARC V9 systems plan for those systems to support both the SPARC V8 ABI interface and the SPARC V9 ABI interface. The generic ABI should be written so that multiple interfaces are clearly permitted.

One issue that arises when supporting multiple interfaces is the need for a different set of shared objects for each interface. In principle, shared object version numbers might be used to distinguish the different interfaces, but in practice this is clumsy and difficult to coordinate. The SPARC International V9 SPARC Compliance Definition (SCD) Special Interest Group (SIG) proposes to use a different subdirectory of each shared object directory for each interface supported. The subdirectory `sparc32` is to be used for SPARC V8; `sparc64` is for SPARC V9. (For backward compatibility, `sparc32` can be a symbolic link to the current directory.)

There are several possible ways the generic ABI could be written so as to clarify that the use of multiple subdirectories is permitted:

1. Make the exact shared object search algorithm processor-specific, as long as the search order is as specified in “Shared Object Dependencies.”
2. Specify that an optional prefix is prepended to the shared object name, but only if the shared object name contains no slashes. The prefix to use is processor-specific.
3. As in (2), but specify an algorithm for computing the prefix. For example, the following algorithm might be used: If the shared object name has no slashes, a machine name string is created by taking the symbolic constant for the value in the `e_machine` field of the ELF header, dropping the `EM_` prefix, and translating all upper-case letters to lower case. For example, the symbolic constant `EM_SPARC64` produces the machine name string `sparc64`. The machine name string, a slash, and the shared object name are concatenated. The resulting string is used to search the three facilities specified above. For example, if `e_machine` is `EM_SPARC32` and the shared object is `lib1`, then the search is for `sparc32/lib1`.

While we do not have exact language to suggest as yet, we seek guidance from the ABICC as to which of the above three approaches is preferable, so that we can then draft and come back with specific language.]

5.4.5. Global Offset Table (Processor-Specific)

[This section is unchanged.]

5.4.6. Procedure Linkage Table (Processor-Specific)

[This section is unchanged.]

5.4.7. Hash Table

[This section is unchanged.]

5.4.8. Initialization and Termination Functions

[This section is unchanged.]

6. LIBRARIES

6.1. Introduction

[Change Figure 6-1 to appear as follows:]

Figure 6-1: Shared Library Names

Library	Reference Name
libc	SEE
libnsl	PROCESSOR
libsys	SPECIFIC
libX	MANUAL

6.2. System Library

6.2.1. Global Data Symbols

`wchar_t _numeric[2];`
This array holds local-specific information, as established by `setlocale` (BA_OS). Specifically, `_numeric[0]` holds the decimal-point character, and `_numeric[1]` holds the character used to separate groups of digits to the left of the decimal-point character in formatted non-monetary quantities. See `localeconv` (BA_LIB) for more information.

6.2.2. Vendor Extensions

Symbols with the prefix _$vendor.*company* provide operating system entries for the vendor named *company*. The system library does not have unadorned alternatives for these names. As an example, the "XYZ Computer Company" might use the prefix _$vendor.xyz.

6.3. C Library

[This section is unchanged.]

6.4. Network Services Library

[Add the following functions[1] to the libnsl library:]

```
xdr_hyper
xdr_u hyper
xdr_u_int      (errata)
xdr_int32
xdr_u_int32
xdr_long_double
```

1. New functions will be explained in the delta document for the SVID.

6.5. X Window System Library

[This section is unchanged.]

6.6. System Data Interfaces

[This section is unchanged.]

7. FORMATS AND PROTOCOLS

7.1. Introduction

[This section is unchanged.]

7.2. Archive File

[This section is unchanged.]

7.3. Other Archive Formats

[We may need to add a 64-bit tape archive format.]

7.4. Terminfo Data Base

[This section is unchanged.]

7.5. Formats and Protocols for Networking

7.5.1. XDR: External Data Representations

[The following text should follow the "Double-precision Floating-point" section.]

Quad-precision Floating-point

XDR defines the encoding for the quad-precision floating-point data type "long double" (128 bits or 16 bytes). The encoding used is a logical extension to the IEEE standard for single and double precision encoding. XDR encodes the following three fields, which describe the quad-precision floating-point number:

S: The sign of the number. Values 0 and 1 represent positive and negative, respectively. One bit.

E: The exponent of the number, base 2. 15 bits are devoted to this field. The exponent is biased by 16383.

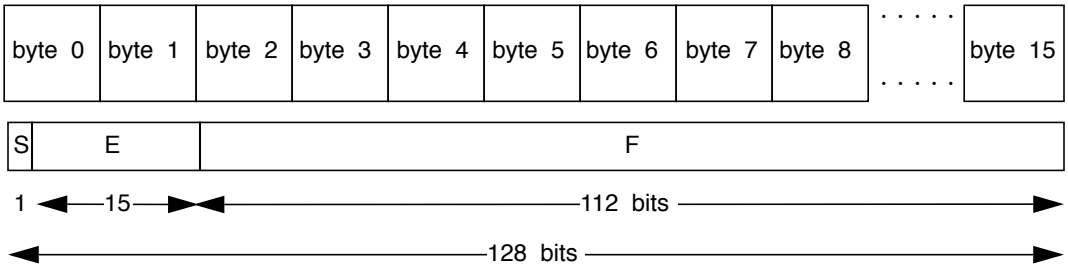
F: The fractional part of the number's mantissa, base 2. 112 bits are devoted to this field.

Therefore, the floating-point number is described by:

```
(-1)**S * 2**(E-Bias) * 1.F
```

It is declared as follows:

*Quad-Precision Floating-poin*t



The most and least significant bits of a quad-precision floating-point number are 0 and 127. The beginning bit (and the most significant bit) offsets of S, E, and F are 0, 1, and 16, respectively. Note that these numbers refer to the mathematical positions of the bits, and NOT to their actual physical locations (which vary from medium to medium).

Even though quad-precision floating-point is not yet a part of the IEEE standard, the IEEE 754 specifications should be consulted concerning the encoding for signed zero, signed infinity (overflow), and denormalized numbers (underflow). According to IEEE specifications, the “NaN” (not a number) is system dependent.

7.5.2. RPC: Remote Procedure Call

[This section is unchanged.]

8. SYSTEM COMMANDS

8.1. Commands for Application Programs

[May need to add a new tape archive command for 64-bits.]

9. EXECUTION ENVIRONMENT

9.1. Application Environment

[This section is unchanged.]

9.2. File System Structure and Contents

[This section is unchanged.]

10. WINDOWING AND TERMINAL INTERFACES

10.1. The System V Window System

[This section is unchanged.]

10.2. System V Window System Components

[This section is unchanged.]

Appendix A. Minor Corrections to Original ABI Supplement

A.1. Formats and Protocols/Terminfo Data Base

[On pages 7-7 and 7-8, the text:]

. . . terminal capabilities are stored here in the order in which **that** are listed under the . . .

[should read:]

. . . terminal capabilities are stored here in the order in which **they** are listed under the . . .

A.2. System Commands/Commands for Application Programs

[On page 8-1 at the bottom of the page, the text:]

. . . UNIX system shell **(shBU_CMD).**

[should read:]

. . . UNIX system shell **sh(BU_CMD).**