
**Linux Standard Base (LSB) core
specification 3.1 —**

**Part 1:
Generic specification**

*Spécifications 3.1 relatives au noyau de base normalisé Linux (LSB) —
Partie 1: Spécifications génériques*



Reference number
ISO/IEC 23360-1:2006(E)

AVA Documents
Document Sharing Hub
© ISO/IEC 2006

PDF disclaimer

This PDF file may contain embedded typefaces. In accordance with Adobe's licensing policy, this file may be printed or viewed but shall not be edited unless the typefaces which are embedded are licensed to and installed on the computer performing the editing. In downloading this file, parties accept therein the responsibility of not infringing Adobe's licensing policy. The ISO Central Secretariat accepts no liability in this area.

Adobe is a trademark of Adobe Systems Incorporated.

Details of the software products used to create this PDF file can be found in the General Info relative to the file; the PDF-creation parameters were optimized for printing. Every care has been taken to ensure that the file is suitable for use by ISO member bodies. In the unlikely event that a problem relating to it is found, please inform the Central Secretariat at the address given below.

© ISO/IEC 2006

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Linux Standard Base Core Specification 3.1

ISO/IEC 23360-1:2006(E)

Copyright © 2006 ISO/IEC

This standard includes material that has been provided by the Free Standards Group under the GNU Free Documentation License Version 1.1 published by the Free Software Foundation.

Portions of the text are copyrighted by the following parties:

- The Regents of the University of California
- Free Software Foundation
- Ian F. Darwin
- Paul Vixie
- BSDI (now Wind River)
- Andrew G Morgan
- Jean-loup Gailly and Mark Adler
- Massachusetts Institute of Technology

These excerpts are being used in accordance with their respective licenses.

Linux is the registered trademark of Linus Torvalds in the U.S. and other countries.

UNIX is a registered trademark of The Open Group.

LSB is a trademark of the Free Standards Group in the United States and other countries.

AMD is a trademark of Advanced Micro Devices, Inc.

Intel and Itanium are registered trademarks and Intel386 is a trademark of Intel Corporation.

PowerPC is a registered trademark and PowerPC Architecture is a trademark of the IBM Corporation.

S/390 is a registered trademark of the IBM Corporation.

OpenGL is a registered trademark of Silicon Graphics, Inc.

Contents

Foreword	viii
Introduction	ix
I Introductory Elements	0
1 Scope.....	1
1.1 General.....	1
1.2 Module Specific Scope.....	1
2 References.....	2
2.1 Normative References	2
2.2 Informative References / Bibliography	3
3 Requirements	6
3.1 Relevant Libraries	6
3.2 LSB Implementation Conformance	6
3.3 LSB Application Conformance.....	7
4 Definitions	9
5 Terminology	10
6 Documentation Conventions	12
7 Relationship To ISO / IEC 9945 POSIX	13
8 Relationship To Other Free Standards Group Specifications	14
II Executable And Linking Format (ELF).....	15
9 Introduction.....	16
10 Low Level System Information.....	17
10.1 Operating System Interface	17
10.2 Machine Interface.....	17
11 Object Format.....	18
11.1 Object Files	18
11.2 Sections	18
11.3 Special Sections.....	21
11.4 Symbol Mapping.....	27
11.5 DWARF Extensions	27
11.6 Exception Frames	30
11.7 Symbol Versioning.....	35
11.8 ABI note tag	39
12 Dynamic Linking	40
12.1 Program Loading and Dynamic Linking.....	40
12.2 Program Header.....	40
12.3 Dynamic Entries	40
III Base Libraries	45
13 Base Libraries	46
13.1 Introduction	46
13.2 Program Interpreter	46
13.3 Interfaces for libc.....	46
13.4 Data Definitions for libc	59
13.5 Interface Definitions for libc	111
13.6 Interfaces for libm	212
13.7 Data Definitions for libm.....	215
13.8 Interface Definitions for libm	222
13.9 Interfaces for libpthread	222
13.10 Data Definitions for libpthread	225
13.11 Interface Definitions for libpthread	229
13.12 Interfaces for libgcc_s	230
13.13 Data Definitions for libgcc_s.....	230
13.14 Interfaces for libdl	231
13.15 Data Definitions for libdl	232

13.16 Interface Definitions for libdl	232
13.17 Interfaces for librt	234
13.18 Interfaces for libcrypt.....	235
13.19 Interfaces for libpam	236
13.20 Data Definitions for libpam	237
13.21 Interface Definitions for libpam	238
IV Utility Libraries.....	250
14 Utility Libraries	251
14.1 Introduction	251
14.2 Interfaces for libz.....	251
14.3 Data Definitions for libz	252
14.4 Interface Definitions for libz.....	254
14.5 Interfaces for libncurses.....	299
14.6 Data Definitions for libncurses.....	304
14.7 Interfaces for libutil.....	312
14.8 Interface Definitions for libutil.....	313
V Commands and Utilities	318
15 Commands and Utilities.....	319
15.1 Commands and Utilities	319
15.2 Command Behavior	320
VI Execution Environment	381
16 File System Hierarchy	382
16.1 /dev: Device Files.....	382
16.2 /etc: Host-specific system configuration	382
16.3 User Accounting Databases.....	384
16.4 Path For System Administration Utilities.....	384
17 Additional Recommendations	385
17.1 Recommendations for applications on ownership and permissions.....	385
18 Additional Behaviors	387
18.1 Mandatory Optional Behaviors.....	387
19 Localization	389
19.1 Introduction	389
19.2 Regular Expressions.....	389
19.3 Pattern Matching Notation	389
VII System Initialization	391
20 System Initialization.....	392
20.1 Cron Jobs	392
20.2 Init Script Actions.....	393
20.3 Comment Conventions for Init Scripts	394
20.4 Installation and Removal of Init Scripts.....	396
20.5 Run Levels.....	397
20.6 Facility Names	398
20.7 Script Names.....	399
20.8 Init Script Functions.....	399
VIII Users & Groups.....	402
21 Users & Groups.....	403
21.1 User and Group Database.....	403
21.2 User & Group Names	403
21.3 User ID Ranges	404
21.4 Rationale.....	404

IX Package Format and Installation.....	405
22 Software Installation	406
22.1 Introduction	406
22.2 Package File Format.....	406
22.3 Package Script Restrictions	425
22.4 Package Tools	425
22.5 Package Naming.....	426
22.6 Package Dependencies	427
22.7 Package Architecture Considerations	427
A Alphabetical Listing of Interfaces.....	428
A.1 libc.....	428
A.2 libcrypt	440
A.3 libdl.....	440
A.4 libm.....	440
A.5 libncurses	445
A.6 libpam	450
A.7 libpthread	450
A.8 librt	452
A.9 libutil	452
A.10 libz	452
B Future Directions (Informative).....	454
B.1 Introduction	454
B.2 Commands And Utilities	455
lsbinstall.....	455

List of Figures

11-1 Version Definition Entries 36

11-2 Version Definition Auxiliary Entries.....37

11-3 Version Needed Entries 37

11-4 Version Needed Auxiliary Entries.....38

12-1 Dynamic Structure.....40

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

International Standard ISO/IEC 23360-1 was prepared by the Free Standards Group and was adopted, under the PAS procedure, by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

ISO/IEC 23360 consists of the following parts, under the general title *Linux Standard Base (LSB) core specification 3.1*:

- *Part 1: Generic specification*
- *Part 2: Specification for IA32 architecture*
- *Part 3: Specification for IA64 architecture*
- *Part 4: Specification for AMD64 architecture*
- *Part 5: Specification for PPC32 architecture*
- *Part 6: Specification for PPC64 architecture*
- *Part 7: Specification for S390 architecture*
- *Part 8: Specification for S390X architecture*

Introduction

The LSB defines a binary interface for application programs that are compiled and packaged for LSB-conforming implementations on many different hardware architectures. Since a binary specification includes information specific to the computer processor architecture for which it is intended, it is not possible for a single document to specify the interface for all possible LSB-conforming implementations. Therefore, the LSB is a family of specifications, rather than a single one.

This document should be used in conjunction with the documents it references. This document enumerates the system components it includes, but descriptions of those components may be included entirely or partly in this document, partly in other documents, or entirely in other reference documents. For example, the section that describes system service routines includes a list of the system routines supported in this interface, formal declarations of the data structures they use that are visible to applications, and a pointer to the underlying referenced specification for information about the syntax and semantics of each call. Only those routines not described in standards referenced by this document, or extensions to those standards, are described in detail. Information referenced in this way is as much a part of this document as is the information explicitly included here.

The specification carries a version number of either the form $x.y$ or $x.y.z$. This version number carries the following meaning:

- The first number (x) is the major version number. All versions with the same major version number should share binary compatibility. Any addition or deletion of a new library results in a new version number. Interfaces marked as deprecated may be removed from the specification at a major version change.
- The second number (y) is the minor version number. Individual interfaces may be added if all certified implementations already had that (previously undocumented) interface. Interfaces may be marked as deprecated at a minor version change. Other minor changes may be permitted at the discretion of the LSB workgroup.
- The third number (z), if present, is the editorial level. Only editorial changes should be included in such versions.

Since this specification is a descriptive Application Binary Interface, and not a source level API specification, it is not possible to make a guarantee of 100% backward compatibility between major releases. However, it is the intent that those parts of the binary interface that are visible in the source level API will remain backward compatible from version to version, except where a feature marked as deprecated in one release may be removed from a future release.

Implementors are strongly encouraged to make use of symbol versioning to permit simultaneous support of applications conforming to different releases of this specification.

This is version 3.1 of the Linux Standard Base Core Specification. This specification is part of a family of specifications under the general title "Linux Standard Base (LSB) core specification 3.1". Developers of applications or implementations interested in using the LSB trademark should see the Free Standards Group Certification Policy for details.

I Introductory Elements

Linux Standard Base (LSB) core specification 3.1 —

Part 1: Generic specification

1 Scope

1.1 General

The Linux Standard Base (LSB) defines a system interface for compiled applications and a minimal environment for support of installation scripts. Its purpose is to enable a uniform industry standard environment for high-volume applications conforming to the LSB.

These specifications are composed of two basic parts: A common specification ("LSB-generic" or "generic LSB"), ISO/IEC 23360-1, describing those parts of the interface that remain constant across all implementations of the LSB, and an architecture-specific part ("LSB-arch" or "archLSB") describing the parts of the interface that vary by processor architecture. Together, the LSB-generic and the relevant architecture-specific part of ISO/IEC 23360 for a single hardware architecture provide a complete interface specification for compiled application programs on systems that share a common hardware architecture.

ISO/IEC 23360-1, the LSB-generic document, should be used in conjunction with an architecture-specific part. Whenever a section of the LSB-generic specification is supplemented by architecture-specific information, the LSB-generic document includes a reference to the architecture part. Architecture-specific parts of ISO/IEC 23360 may also contain additional information that is not referenced in the LSB-generic document.

The LSB contains both a set of Application Program Interfaces (APIs) and Application Binary Interfaces (ABIs). APIs may appear in the source code of portable applications, while the compiled binary of that application may use the larger set of ABIs. A conforming implementation provides all of the ABIs listed here. The compilation system may replace (e.g. by macro definition) certain APIs with calls to one or more of the underlying binary interfaces, and may insert calls to binary interfaces as needed.

The LSB is primarily a binary interface definition. Not all of the source level APIs available to applications may be contained in this specification.

1.2 Module Specific Scope

This is the Core module of the Linux Standard Base (LSB), ISO/IEC 23360-1. This module provides the fundamental system interfaces, libraries, and runtime environment upon which all conforming applications and libraries depend.

Interfaces described in this part of ISO/IEC 23360 are mandatory except where explicitly listed otherwise. Core interfaces may be supplemented by other modules; all modules are built upon the core.

2 References

2.1 Normative References

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

Note: Where copies of a document are available on the World Wide Web, a Uniform Resource Locator (URL) is given for informative purposes only. This may point to a more recent copy of the referenced specification, or may be out of date. Reference copies of specifications at the revision level indicated may be found at the Free Standards Group's Reference Specifications (<http://refspecs.freestandards.org>) site.

Table 2-1 Normative References

Name	Title	URL
Filesystem Hierarchy Standard	Filesystem Hierarchy Standard (FHS) 2.3	http://www.pathname.com/fhs/
ISO C (1999)	ISO / IEC 9899: 1999, <i>Programming Languages — C</i>	
ISO POSIX (2003)	ISO / IEC 9945-1:2003, <i>Information technology — Portable Operating System Interface (POSIX) — Part 1: Base Definitions</i> ISO / IEC 9945-2:2003, <i>Information technology — Portable Operating System Interface (POSIX) — Part 2: System Interfaces</i> ISO / IEC 9945-3:2003, <i>Information technology — Portable Operating System Interface (POSIX) — Part 3: Shell and Utilities</i> ISO / IEC 9945-4:2003, <i>Information technology — Portable Operating System Interface (POSIX) — Part 4: Rationale</i>	http://www.unix.org/version3/

Name	Title	URL
Itanium™ C++ ABI	Itanium™ C++ ABI (Revision 1.83)	http://refspecs.freestdards.org/cxxabi-1.83.html
Large File Support	Large File Support	http://www.UNIX-systems.org/version2/whatsnew/lfs20mar.html
SUSv2	CAE Specification, January 1997, System Interfaces and Headers (XSH), Issue 5 (ISBN: 1-85912-181-0, C606)	http://www.opengroup.org/publications/catalog/un.htm
SVID Issue 3	American Telephone and Telegraph Company, System V Interface Definition, Issue 3; Morristown, NJ, UNIX Press, 1989. (ISBN 0201566524)	
SVID Issue 4	System V Interface Definition, Fourth Edition	
System V ABI	System V Application Binary Interface, Edition 4.1	http://www.caldera.com/developers/devspecs/gabi41.pdf
System V ABI Update	System V Application Binary Interface - DRAFT - 17 December 2003	http://www.caldera.com/developers/gabi/2003-12-17/contents.html
X/Open Curses	CAE Specification, May 1996, X/Open Curses, Issue 4, Version 2 (ISBN: 1-85912-171-3, C610), plus Corrigendum U018	http://www.opengroup.org/publications/catalog/un.htm

2.2 Informative References/Bibliography

In addition, the specifications listed below provide essential background information to implementors of this specification. These references are included for information only.

Table 2-2 Other References

Name	Title	URL
DWARF Debugging Information Format, Revision 2.0.0	DWARF Debugging Information Format, Revision 2.0.0 (July 27, 1993)	http://refspecs.freestdards.org/dwarf/dwarf-2.0.0.pdf
DWARF Debugging Information Format, Revision 3.0.0 (Draft)	DWARF Debugging Information Format, Revision 3.0.0 (Draft)	http://refspecs.freestdards.org/dwarf/
IEC 60559 / IEEE 754 Floating Point	IEC 60559:1989, <i>Binary floating-point arithmetic for microprocessor systems</i>	http://www.ieee.org/
ISO / IEC TR 14652	ISO / IEC TR 14652:2004, <i>Information technology — Specification method for cultural conventions</i>	
ITU-T V.42	International Telecommunication Union Recommendation V.42 (2002): Error-correcting procedures for DCEs using asynchronous-to-synchronous conversion ITUV	http://www.itu.int/rec/recommendation.asp?type=folders&lang=e&parent=T-REC-V.42
Li18nux Globalization Specification	LI18NIX 2000 Globalization Specification, Version 1.0 with Amendment 4	http://www.li18nux.org/docs/html/LI18NIX-2000-amd4.htm
Linux Allocated Device Registry	LINUX ALLOCATED DEVICES	http://www.lanana.org/docs/device-list/devices.txt
PAM	Open Software Foundation, Request For Comments: 86.0, October 1995, V. Samar & R.Schemers (SunSoft)	http://www.opengroup.org/tech/rfc/mirror-rfc/rfc86.0.txt

Name	Title	URL
RFC 1321: The MD5 Message-Digest Algorithm	IETF RFC 1321: The MD5 Message-Digest Algorithm	http://www.ietf.org/rfc/rfc1321.txt
RFC 1831 / 1832 RPC & XDR	IETF RFC 1831 & 1832	http://www.ietf.org/
RFC 1833: Binding Protocols for ONC RPC Version 2	IETF RFC 1833: Binding Protocols for ONC RPC Version 2	http://www.ietf.org/rfc/rfc1833.txt
RFC 1950: ZLIB Compressed Data Format Specification	IETF RFC 1950: ZLIB Compressed Data Format Specification	http://www.ietf.org/rfc/rfc1950.txt
RFC 1951: DEFLATE Compressed Data Format Specification	IETF RFC 1951: DEFLATE Compressed Data Format Specification version 1.3	http://www.ietf.org/rfc/rfc1951.txt
RFC 1952: GZIP File Format Specification	IETF RFC 1952: GZIP file format specification version 4.3	http://www.ietf.org/rfc/rfc1952.txt
RFC 2440: OpenPGP Message Format	IETF RFC 2440: OpenPGP Message Format	http://www.ietf.org/rfc/rfc2440.txt
RFC 2821: Simple Mail Transfer Protocol	IETF RFC 2821: Simple Mail Transfer Protocol	http://www.ietf.org/rfc/rfc2821.txt
RFC 2822: Internet Message Format	IETF RFC 2822: Internet Message Format	http://www.ietf.org/rfc/rfc2822.txt
RFC 791: Internet Protocol	IETF RFC 791: Internet Protocol Specification	http://www.ietf.org/rfc/rfc791.txt
RPM Package Format	RPM Package Format V3.0	http://www.rpm.org/max-rpm/s1-rpm-file-format-rpm-file-format.html
SUSv2 Commands and Utilities	The Single UNIX Specification(SUS) Version 2, Commands and Utilities (XCU), Issue 5 (ISBN: 1-85912-191-8, C604)	http://www.opengroup.org/publications/catalog/un.htm
zlib Manual	zlib 1.2 Manual	http://www.gzip.org/zlib/

3 Requirements

3.1 Relevant Libraries

The libraries listed in Table 3-1 shall be available on a Linux Standard Base system, with the specified runtime names. The libraries listed in Table 3-2 are architecture specific, but shall be available on all LSB conforming systems. This list may be supplemented or amended by the relevant architecture specific part of ISO / IEC 23360.

Table 3-1 Standard Library Names

Library	Runtime Name
libdl	libdl.so.2
libcrypt	libcrypt.so.1
libz	libz.so.1
libncurses	libncurses.so.5
libutil	libutil.so.1
libpthread	libpthread.so.0
librt	librt.so.1
libpam	libpam.so.0
libgcc_s	libgcc_s.so.1

Table 3-2 Standard Library Names defined in the Architecture Specific Parts of ISO/IEC 23360

Library	Runtime Name
libm	See archLSB
libc	See archLSB
proginterp	See archLSB

These libraries will be in an implementation-defined directory which the dynamic linker shall search by default.

3.2 LSB Implementation Conformance

A conforming implementation is necessarily architecture specific, and must provide the interfaces specified by both the generic LSB Core specification (ISO / IEC 23360-1) and the relevant architecture specific part of ISO / IEC 23360.

Rationale: An implementation must provide *at least* the interfaces specified in these specifications. It may also provide additional interfaces.

A conforming implementation shall satisfy the following requirements:

- A processor architecture represents a family of related processors which may not have identical feature sets. The architecture specific parts of ISO / IEC 23360 that supplement this specification for a given target processor architecture describe a minimum acceptable processor. The implementation

shall provide all features of this processor, whether in hardware or through emulation transparent to the application.

- The implementation shall be capable of executing compiled applications having the format and using the system interfaces described in this document.
- The implementation shall provide libraries containing the interfaces specified by this document, and shall provide a dynamic linking mechanism that allows these interfaces to be attached to applications at runtime. All the interfaces shall behave as specified in this document.
- The map of virtual memory provided by the implementation shall conform to the requirements of this document.
- The implementation's low-level behavior with respect to function call linkage, system traps, signals, and other such activities shall conform to the formats described in this document.
- The implementation shall provide all of the mandatory interfaces in their entirety.
- The implementation may provide one or more of the optional interfaces. Each optional interface that is provided shall be provided in its entirety. The product documentation shall state which optional interfaces are provided.
- The implementation shall provide all files and utilities specified as part of this document in the format defined here and in other referenced documents. All commands and utilities shall behave as required by this document. The implementation shall also provide all mandatory components of an application's runtime environment that are included or referenced in this document.
- The implementation, when provided with standard data formats and values at a named interface, shall provide the behavior defined for those values and data formats at that interface. However, a conforming implementation may consist of components which are separately packaged and/or sold. For example, a vendor of a conforming implementation might sell the hardware, operating system, and windowing system as separately packaged items.
- The implementation may provide additional interfaces with different names. It may also provide additional behavior corresponding to data values outside the standard ranges, for standard named interfaces.

3.3 LSB Application Conformance

A conforming application is necessarily architecture specific, and must conform to both the generic LSB Core specification (ISO/IEC 23360-1) and the relevant architecture specific part of ISO/IEC 23360.

A conforming application shall satisfy the following requirements:

- Its executable files shall be either shell scripts or object files in the format defined for the Object File Format system interface.
- Its object files shall participate in dynamic linking as defined in the Program Loading and Linking System interface.
- It shall employ only the instructions, traps, and other low-level facilities defined in the Low-Level System interface as being for use by applications.
- If it requires any optional interface defined in this document in order to be installed or to execute successfully, the requirement for that optional interface shall be stated in the application's documentation.

- It shall not use any interface or data format that is not required to be provided by a conforming implementation, unless:
 - If such an interface or data format is supplied by another application through direct invocation of that application during execution, that application shall be in turn an LSB conforming application.
 - The use of that interface or data format, as well as its source, shall be identified in the documentation of the application.
- It shall not use any values for a named interface that are reserved for vendor extensions.

A strictly conforming application shall not require or use any interface, facility, or implementation-defined extension that is not defined in this document in order to be installed or to execute successfully.

4 Definitions

For the purposes of this document, the following definitions, as specified in the *ISO/IEC Directives, Part 2, 2004, 5th Edition*, apply:

can

be able to; there is a possibility of; it is possible to

cannot

be unable to; there is no possibility of; it is not possible to

may

is permitted; is allowed; is permissible

need not

it is not required that; no...is required

shall

is to; is required to; it is required that; has to; only...is permitted; it is necessary

shall not

is not allowed [permitted] [acceptable] [permissible]; is required to be not; is required that...be not; is not to be

should

it is recommended that; ought to

should not

it is not recommended that; ought not to

5 Terminology

For the purposes of this document, the following terms apply:

archLSB

The architectural part of the LSB Specification which describes the specific parts of the interface that are platform specific. The archLSB is complementary to the gLSB.

Binary Standard

The total set of interfaces that are available to be used in the compiled binary code of a conforming application.

gLSB

The common part of the LSB Specification that describes those parts of the interface that remain constant across all hardware implementations of the LSB.

implementation-defined

Describes a value or behavior that is not defined by this document but is selected by an implementor. The value or behavior may vary among implementations that conform to this document. An application should not rely on the existence of the value or behavior. An application that relies on such a value or behavior cannot be assured to be portable across conforming implementations. The implementor shall document such a value or behavior so that it can be used correctly by an application.

Shell Script

A file that is read by an interpreter (e.g., awk). The first line of the shell script includes a reference to its interpreter binary.

Source Standard

The set of interfaces that are available to be used in the source code of a conforming application.

undefined

Describes the nature of a value or behavior not defined by this document which results from use of an invalid program construct or invalid data input. The value or behavior may vary among implementations that conform to this document. An application should not rely on the existence or validity of the value or behavior. An application that relies on any particular value or behavior cannot be assured to be portable across conforming implementations.

unspecified

Describes the nature of a value or behavior not specified by this document which results from use of a valid program construct or valid data input. The value or behavior may vary among implementations that conform to this document. An application should not rely on the existence or validity of the value or behavior. An application that relies on any particular value or behavior cannot be assured to be portable across conforming implementations.

Other terms and definitions used in this document shall have the same meaning as defined in Chapter 3 of the Base Definitions volume of [ISO POSIX \(2003\)](#).

6 Documentation Conventions

Throughout this document, the following typographic conventions are used:

`function()`

the name of a function

command

the name of a command or utility

CONSTANT

a constant value

parameter

a parameter

variable

a variable

Throughout this specification, several tables of interfaces are presented. Each entry in these tables has the following format:

name

the name of the interface

(symver)

An optional symbol version identifier, if required.

[refno]

A reference number indexing the table of referenced specifications that follows this table.

For example,

forkpty(GLIBC_2.0) [SUSv3]

refers to the interface named `forkpty()` with symbol version `GLIBC_2.0` that is defined in the `SUSv3` reference.

Note: Symbol versions are defined in the architecture specific parts of ISO/IEC 23360 only.

7 Relationship To ISO/IEC 9945 POSIX

This specification includes many interfaces described in [ISO POSIX \(2003\)](#). Unless otherwise specified, such interfaces should behave exactly as described in that specification. Any conflict between the requirements described here and the [ISO POSIX \(2003\)](#) standard is unintentional, except as explicitly noted otherwise.

Note: In addition to the differences noted inline in this specification, PDTR 24715 has extracted the differences between this specification and [ISO POSIX \(2003\)](#) into a single place. It is the long term plan of the Free Standards Group to converge the LSB Core Specification with ISO / IEC 9945 POSIX.

The LSB Specification Authority is responsible for deciding the meaning of conformance to normative referenced standards in the LSB context. Problem Reports regarding underlying or referenced standards in any other context will be referred to the relevant maintenance body for that standard.

8 Relationship To Other Free Standards Group Specifications

The LSB is the base for several other specification projects under the umbrella of the Free Standards Group (FSG). This specification is the foundation, and other specifications build on the interfaces defined here. However, beyond those specifications listed as Normative References, this specification has no dependencies on other FSG projects.

II Executable And Linking Format (ELF)

9 Introduction

Executable and Linking Format (ELF) defines the object format for compiled applications. This specification supplements the information found in [System V ABI Update](#) and is intended to document additions made since the publication of that document.

10 Low Level System Information

10.1 Operating System Interface

LSB-conforming applications shall assume that stack, heap and other allocated memory regions will be non-executable. The application must take steps to make them executable if needed.

10.2 Machine Interface

10.2.1 Data Representation

LSB-conforming applications shall use the data representation as defined in the Architecture specific ELF documents.

10.2.1.1 Fundamental Types

In addition to the fundamental types specified in the relevant architecture specific part of ISO/IEC 23360, a 1 byte data type is defined here.

Table 10-1 Scalar Types

Type	C	C++	sizeof	Align- ment (bytes)	Architec- ture Rep- resenta- tion
Integral	_Bool	bool	1	1	byte

11 Object Format

11.1 Object Files

LSB-conforming implementations shall support the object file Executable and Linking Format (ELF), which is defined by the following documents:

- [System V ABI](#)
- [System V ABI Update](#)
- this specification
- the relevant architecture specific part of ISO / IEC 23360

Conforming implementations may also support other unspecified object file formats.

11.2 Sections

11.2.1 Introduction

As described in [System V ABI](#), an ELF object file contains a number of *sections*.

11.2.2 Sections Types

The section header table is an array of Elf32_Shdr or Elf64_Shdr structures as described in [System V ABI](#). The *sh_type* member shall be either a value from Table 11-1, drawn from the System V ABI, or one of the additional values specified in Table 11-2.

A section header's *sh_type* member specifies the sections's semantics.

11.2.2.1 ELF Section Types

The following section types are defined in the [System V ABI](#) and the [System V ABI Update](#).

Table 11-1 ELF Section Types

Name	Value	Description
SHT_DYNAMIC	0x6	The section holds information for dynamic linking. Currently, an object file shall have only one dynamic section, but this restriction may be relaxed in the future. See 'Dynamic Section' in Chapter 5 for details.
SHT_DYNSYM	0xb	This section holds a minimal set of symbols adequate for dynamic linking. See also SHT_SYMTAB. Currently, an object file may have either a sec-

Name	Value	Description
		tion of SHT_SYMTAB type or a section of SHT_DYNSYM type, but not both. This restriction may be relaxed in the future.
SHT_FINI_ARRAY	0xf	This section contains an array of pointers to termination functions, as described in 'Initialization and Termination Functions' in Chapter 5. Each pointer in the array is taken as a parameterless procedure with a void return.
SHT_HASH	0x5	The section holds a symbol hash table. Currently, an object file shall have only one hash table, but this restriction may be relaxed in the future. See 'Hash Table' in the Chapter 5 for details.
SHT_INIT_ARRAY	0xe	This section contains an array of pointers to initialization functions, as described in 'Initialization and Termination Functions' in Chapter 5. Each pointer in the array is taken as a parameterless procedure with a void return.
SHT_NOBITS	0x8	A section of this type occupies no space in the file but otherwise resembles SHT_PROGBITS. Although this section contains no bytes, the sh_offset member contains the conceptual file offset.
SHT_NOTE	0x7	The section holds information that marks the file in some way. See 'Note Section' in Chapter 5 for details.

Name	Value	Description
SHT_NULL	0x0	This value marks the section header as inactive; it does not have an associated section. Other members of the section header have undefined values.
SHT_PREINIT_ARRAY	0x10	This section contains an array of pointers to functions that are invoked before all other initialization functions, as described in 'Initialization and Termination Functions' in Chapter 5. Each pointer in the array is taken as a parameterless procedure with a void return.
SHT_PROGBITS	0x1	The section holds information defined by the program, whose format and meaning are determined solely by the program.
SHT_REL	0x9	The section holds relocation entries without explicit addends, such as type Elf32_Rel for the 32-bit class of object files or type Elf64_Rel for the 64-bit class of object files. An object file may have multiple relocation sections. See "Relocation"
SHT_RELA	0x4	The section holds relocation entries with explicit addends, such as type Elf32_Rela for the 32-bit class of object files or type Elf64_Rela for the 64-bit class of object files. An object file may have multiple relocation sections. 'Relocation' b
SHT_STRTAB	0x3	The section holds a string table. An object file may have multiple

Name	Value	Description
		string table sections. See 'String Table' below for details.
SHT_SYMTAB	0x2	This section holds a symbol table. Currently, an object file may have either a section of SHT_SYMTAB type or a section of SHT_DYNSYM type, but not both. This restriction may be relaxed in the future. Typically, SHT_SYMTAB provides symbols for link editing, though it may also be used for dynamic linking. As a complete symbol table, it may contain many symbols unnecessary for dynamic linking.

11.2.2.2 Additional Section Types

The following additional section types are defined here.

Table 11-2 Additional Section Types

Name	Value	Description
SHT_GNU_verdef	0x6ffffffd	This section contains the symbol versions that are provided.
SHT_GNU_verneed	0x6ffffffe	This section contains the symbol versions that are required.
SHT_GNU_versym	0x6fffffff	This section contains the Symbol Version Table.

11.3 Special Sections

11.3.1 Special Sections

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

11.3.1.1 ELF Special Sections

The following sections are defined in the [System V ABI](#) and the [System V ABI Update](#).

Table 11-3 ELF Special Sections

Name	Type	Attributes
.bss	SHT_NOBITS	SHF_ALLOC+SHF_WRITE
.comment	SHT_PROGBITS	0
.data	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE
.data1	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE
.debug	SHT_PROGBITS	0
.dynamic	SHT_DYNAMIC	SHF_ALLOC+SHF_WRITE
.dynstr	SHT_STRTAB	SHF_ALLOC
.dynsym	SHT_DYNSYM	SHF_ALLOC
.fini	SHT_PROGBITS	SHF_ALLOC+SHF_EXECINSTR
.fini_array	SHT_FINI_ARRAY	SHF_ALLOC+SHF_WRITE
.hash	SHT_HASH	SHF_ALLOC
.init	SHT_PROGBITS	SHF_ALLOC+SHF_EXECINSTR
.init_array	SHT_INIT_ARRAY	SHF_ALLOC+SHF_WRITE
.interp	SHT_PROGBITS	SHF_ALLOC
.line	SHT_PROGBITS	0
.note	SHT_NOTE	0
.preinit_array	SHT_PREINIT_ARRAY	SHF_ALLOC+SHF_WRITE
.rodata	SHT_PROGBITS	SHF_ALLOC
.rodata1	SHT_PROGBITS	SHF_ALLOC
.shstrtab	SHT_STRTAB	0
.strtab	SHT_STRTAB	SHF_ALLOC
.symtab	SHT_SYMTAB	SHF_ALLOC
.tbss	SHT_NOBITS	SHF_ALLOC+SHF_WRITE+SHF_TLS
.tdata	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE+SHF_TLS
.text	SHT_PROGBITS	SHF_ALLOC+SHF_EXEC

Name	Type	Attributes
		ECINSTR

.bss

This section holds data that contributes to the program's memory image. The program may treat this data as uninitialized. However, the system shall initialize this data with zeroes when the program begins to run. The section occupies no file space, as indicated by the section type, SHT_NOBITS

.comment

This section holds version control information.

.data

This section holds initialized data that contribute to the program's memory image.

.data1

This section holds initialized data that contribute to the program's memory image.

.debug

This section holds information for symbolic debugging. The contents are unspecified. All section names with the prefix `.debug` hold information for symbolic debugging. The contents of these sections are unspecified.

.dynamic

This section holds dynamic linking information. The section's attributes will include the SHF_ALLOC bit. Whether the SHF_WRITE bit is set is processor specific. See Chapter 5 for more information.

.dynstr

This section holds strings needed for dynamic linking, most commonly the strings that represent the names associated with symbol table entries. See Chapter 5 for more information.

.dynsym

This section holds the dynamic linking symbol table, as described in 'Symbol Table'. See Chapter 5 for more information.

.fini

This section holds executable instructions that contribute to the process termination code. That is, when a program exits normally, the system arranges to execute the code in this section.

.fini_array

This section holds an array of function pointers that contributes to a single termination array for the executable or shared object containing the section.

.hash

This section holds a symbol hash table. See 'Hash Table' in Chapter 5 for more information.

.init

This section holds executable instructions that contribute to the process initialization code. When a program starts to run, the system arranges to execute the code in this section before calling the main program entry point (called main for C programs)

.init_array

This section holds an array of function pointers that contributes to a single initialization array for the executable or shared object containing the section.

.interp

This section holds the path name of a program interpreter. If the file has a loadable segment that includes relocation, the sections' attributes will include the SHF_ALLOC bit; otherwise, that bit will be off. See Chapter 5 for more information.

.line

This section holds line number information for symbolic debugging, which describes the correspondence between the source program and the machine code. The contents are unspecified.

.note

This section holds information in the format that 'Note Section' in Chapter 5 describes of the System V Application Binary Interface, Edition 4.1.

.preinit_array

This section holds an array of function pointers that contributes to a single pre-initialization array for the executable or shared object containing the section.

.rodata

This section holds read-only data that typically contribute to a non-writable segment in the process image. See 'Program Header' in Chapter 5 for more information.

.rodata1

This section hold sread-only data that typically contribute to a non-writable segment in the process image. See 'Program Header' in Chapter 5 for more information.

.shstrtab

This section holds section names.

.strtab

This section holds strings, most commonly the strings that represent the names associated with symbol table entries. If the file has a loadable segment that includes the symbol string table, the section's attributes will include the SHF_ALLOC bit; otherwi

.symtab

This section holds a symbol table, as 'Symbol Table' in this chapter describes. If the file has a loadable segment that includes the symbol table,

the section's attributes will include the SHF_ALLOC bit; otherwise, that bit will be off.

.tbss

This section holds uninitialized thread-local data that contribute to the program's memory image. By definition, the system initializes the data with zeros when the data is instantiated for each new execution flow. The section occupies no file space, as indicated by the section type, SHT_NOBITS. Implementations need not support thread-local storage.

.tdata

This section holds initialized thread-local data that contributes to the program's memory image. A copy of its contents is instantiated by the system for each new execution flow. Implementations need not support thread-local storage.

.text

This section holds the `text,' or executable instructions, of a program.

11.3.1.2 Additional Special Sections

Object files in an LSB conforming application may also contain one or more of the additional special sections described below.

Table 11-4 Additional Special Sections

Name	Type	Attributes
.ctors	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE
.data.rel.ro	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE
.dtors	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE
.eh_frame	SHT_PROGBITS	SHF_ALLOC
.eh_frame_hdr	SHT_PROGBITS	SHF_ALLOC
.gcc_except_table	SHT_PROGBITS	SHF_ALLOC
.gnu.version	SHT_GNU_versym	SHF_ALLOC
.gnu.version_d	SHT_GNU_verdef	SHF_ALLOC
.gnu.version_r	SHT_GNU_verneed	SHF_ALLOC
.got.plt	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE
.jcr	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE
.note.ABI-tag	SHT_NOTE	SHF_ALLOC
.stab	SHT_PROGBITS	0
.stabstr	SHT_STRTAB	0

`.ctors`

This section contains a list of global constructor function pointers.

`.data.rel.ro`

This section holds initialized data that contribute to the program's memory image. This section may be made read-only after relocations have been applied.

`.dtors`

This section contains a list of global destructor function pointers.

`.eh_frame`

This section contains information necessary for frame unwinding during exception handling. See Section 11.6.1.

`.eh_frame_hdr`

This section contains a pointer to the `.eh_frame` section which is accessible to the runtime support code of a C++ application. This section may also contain a binary search table which may be used by the runtime support code to more efficiently access records in the `.eh_frame` section. See Section 11.6.2.

`.gcc_except_table`

This section holds Language Specific Data.

`.gnu.version`

This section contains the Symbol Version Table. See Section 11.7.2.

`.gnu.version_d`

This section contains the Version Definitions. See Section 11.7.3.

`.gnu.version_r`

This section contains the Version Requirements. See Section 11.7.4.

`.got.plt`

This section holds the read-only portion of the Global Offset Table. This section may be made read-only after relocations have been applied.

`.jcr`

This section contains information necessary for registering compiled Java classes. The contents are compiler-specific and used by compiler initialization functions.

`.note.ABI-tag`

Specify ABI details. See Section 11.8.

`.stab`

This section contains debugging information. The contents are not specified as part of the LSB.

.stabstr

This section contains strings associated with the debugging information contained in the .stab section.

11.4 Symbol Mapping

11.4.1 Introduction

Symbols in a source program are translated by the compilation system into symbols that exist in the object file.

11.4.1.1 C Language

External C symbols shall be unchanged in an object file's symbol table.

11.5 DWARF Extensions

The LSB does not specify debugging information, however, some additional sections contain information which is encoded using the the encoding as specified by [DWARF Debugging Information Format, Revision 2.0.0](#) with extensions defined here.

Note: The extensions specified here also exist in [DWARF Debugging Information Format, Revision 3.0.0 \(Draft\)](#). It is expected that future versions of the LSB will reference the final version of that document, and that the definitions here will be taken from that document instead of being specified here.

11.5.1 DWARF Exception Header Encoding

The DWARF Exception Header Encoding is used to describe the type of data used in the .eh_frame and .eh_frame_hdr section. The upper 4 bits indicate how the value is to be applied. The lower 4 bits indicate the format of the data.

Table 11-5 DWARF Exception Header value format

Name	Value	Meaning
DW_EH_PE_absptr	0x00	The Value is a literal pointer whose size is determined by the architecture.
DW_EH_PE_uleb128	0x01	Unsigned value is encoded using the Little Endian Base 128 (LEB128) as defined by DWARF Debugging Information Format, Revision 2.0.0 .
DW_EH_PE_ushort	0x02	A 2 bytes unsigned value.
DW_EH_PE_ulong	0x03	A 4 bytes unsigned value.
DW_EH_PE_udata8	0x04	An 8 bytes unsigned value.
DW_EH_PE_sleb128	0x09	Signed value is encoded

Name	Value	Meaning
		using the Little Endian Base 128 (LEB128) as defined by DWARF Debugging Information Format, Revision 2.0.0 .
DW_EH_PE_sdata2	0x0A	A 2 bytes signed value.
DW_EH_PE_sdata4	0x0B	A 4 bytes signed value.
DW_EH_PE_sdata8	0x0C	An 8 bytes signed value.

Table 11-6 DWARF Exception Header application

Name	Value	Meaning
DW_EH_PE_pcrel	0x10	Value is relative to the current program counter.
DW_EH_PE_textrel	0x20	Value is relative to the beginning of the .text section.
DW_EH_PE_datarel	0x30	Value is relative to the beginning of the .got or .eh_frame_hdr section.
DW_EH_PE_funcrel	0x40	Value is relative to the beginning of the function.
DW_EH_PE_aligned	0x50	Value is aligned to an address unit sized boundary.

One special encoding, 0xff (DW_EH_PE_omit), shall be used to indicate that no value is present.

11.5.2 DWARF CFI Extensions

In addition to the Call Frame Instructions defined in section 6.4.2 of [DWARF Debugging Information Format, Revision 2.0.0](#), the following additional Call Frame Instructions may also be used.

Table 11-7 Additional DWARF Call Frame Instructions

Name	Value	Meaning
DW_CFA_expression	0x10	The DW_CFA_expression instruction takes two operands: an unsigned LEB128 value representing a register number, and a DW_FORM_block

Name	Value	Meaning
		value representing a DWARF expression. The required action is to establish the DWARF expression as the means by which the address in which the given register contents are found may be computed. The value of the CFA is pushed on the DWARF evaluation stack prior to execution of the DWARF expression. The DW_OP_call2, DW_OP_call4, DW_OP_call_ref and DW_OP_push_object_address DWARF operators (see Section 2.4.1 of DWARF Debugging Information Format, Revision 2.0.0) cannot be used in such a DWARF expression.
DW_CFA_offset_extended_sf	0x11	The DW_CFA_offset_extended_sf instruction takes two operands: an unsigned LEB128 value representing a register number and a signed LEB128 factored offset. This instruction is identical to DW_CFA_offset_extended except that the second operand is signed.
DW_CFA_def_cfa_sf	0x12	The DW_CFA_def_cfa_sf instruction takes two operands: an unsigned LEB128 value representing a register number and a signed LEB128 factored offset. This instruction is identical to DW_CFA_def_cfa except that the second

Name	Value	Meaning
		operand is signed and factored.
DW_CFA_def_cfa_offset_sf	0x13	The DW_CFA_def_cfa_offset_sf instruction takes a signed LEB128 operand representing a factored offset. This instruction is identical to DW_CFA_def_cfa_offset except that the operand is signed and factored.
DW_CFA_GNU_args_size	0x2e	The DW_CFA_GNU_args_size instruction takes an unsigned LEB128 operand representing an argument size. This instruction specifies the total of the size of the arguments which have been pushed onto the stack.
DW_CFA_GNU_negative_offset_extended	0x2f	The DW_CFA_def_cfa_sf instruction takes two operands: an unsigned LEB128 value representing a register number and an unsigned LEB128 which represents the magnitude of the offset. This instruction is identical to DW_CFA_offset_extended_sf except that the operand is subtracted to produce the offset. This instructions is obsoleted by DW_CFA_offset_extended_sf.

11.6 Exception Frames

When using languages that support exceptions, such as C++, additional information must be provided to the runtime environment that describes the call frames that must be unwound during the processing of an exception. This information is contained in the special sections `.eh_frame` and `.eh_framehdr`.

Note: The format of the `.eh_frame` section is similar in format and purpose to the `.debug_frame` section which is specified in [DWARF Debugging Information Format, Revision 3.0.0 \(Draft\)](#). Readers are advised that there are some subtle difference, and care should be taken when comparing the two sections.

11.6.1 The `.eh_frame` section

The `.eh_frame` section shall contain 1 or more Call Frame Information (CFI) records. The number of records present shall be determined by size of the section as contained in the section header. Each CFI record contains a Common Information Entry (CIE) record followed by 1 or more Frame Description Entry (FDE) records. Both CIEs and FDEs shall be aligned to an addressing unit sized boundary.

Table 11-8 Call Frame Information Format

Common Information Entry Record
Frame Description Entry Record(s)

11.6.1.1 The Common Information Entry Format

Table 11-9 Common Information Entry Format

Length	Required
Extended Length	Optional
CIE ID	Required
Version	Required
Augmentation String	Required
Code Alignment Factor	Required
Data Alignment Factor	Required
Return Address Register	Required
Augmentation Data Length	Optional
Augmentation Data	Optional
Initial Instructions	Required
Padding	

Length

A 4 byte unsigned value indicating the length in bytes of the CIE structure, not including the *Length* field itself. If *Length* contains the value 0xffffffff, then the length is contained in the *Extended Length* field. If *Length* contains the value 0, then this CIE shall be considered a terminator and processing shall end.

Extended Length

A 8 byte unsigned value indicating the length in bytes of the CIE structure, not including the *Length* and *Extended Length* fields.

CIE ID

A 4 byte unsigned value that is used to distinguish CIE records from FDE records. This value shall always be 0, which indicates this record is a CIE.

Version

A 1 byte value that identifies the version number of the frame information structure. This value shall be 1.

Augmentation String

This value is a NUL terminated string that identifies the augmentation to the CIE or to the FDEs associated with this CIE. A zero length string indicates that no augmentation data is present. The augmentation string is case sensitive and shall be interpreted as described below.

Code Alignment Factor

An unsigned LEB128 encoded value that is factored out of all advance location instructions that are associated with this CIE or its FDEs. This value shall be multiplied by the delta argument of an advance location instruction to obtain the new location value.

Data Alignment Factor

A signed LEB128 encoded value that is factored out of all offset instructions that are associated with this CIE or its FDEs. This value shall be multiplied by the register offset argument of an offset instruction to obtain the new offset value.

Augmentation Length

An unsigned LEB128 encoded value indicating the length in bytes of the Augmentation Data. This field is only present if the Augmentation String contains the character 'z'.

Augmentation Data

A block of data whose contents are defined by the contents of the Augmentation String as described below. This field is only present if the Augmentation String contains the character 'z'. The size of this data is given by the Augmentation Length.

Initial Instructions

Initial set of Call Frame Instructions. The number of instructions is determined by the remaining space in the CIE record.

Padding

Extra bytes to align the CIE structure to an addressing unit size boundary.

11.6.1.1.1 Augmentation String Format

The Augmentation String indicates the presence of some optional fields, and how those fields should be interpreted. This string is case sensitive. Each character in the augmentation string in the CIE can be interpreted as below:

'z'

A 'z' may be present as the first character of the string. If present, the Augmentation Data field shall be present. The contents of the

Augmentation Data shall be interpreted according to other characters in the Augmentation String.

'L'

A 'L' may be present at any position after the first character of the string. This character may only be present if 'z' is the first character of the string. If present, it indicates the presence of one argument in the Augmentation Data of the CIE, and a corresponding argument in the Augmentation Data of the FDE. The argument in the Augmentation Data of the CIE is 1-byte and represents the pointer encoding used for the argument in the Augmentation Data of the FDE, which is the address of a language-specific data area (LSDA). The size of the LSDA pointer is specified by the pointer encoding used.

'P'

A 'P' may be present at any position after the first character of the string. This character may only be present if 'z' is the first character of the string. If present, it indicates the presence of two arguments in the Augmentation Data of the CIE. The first argument is 1-byte and represents the pointer encoding used for the second argument, which is the address of a *personality routine* handler. The personality routine is used to handle language and vendor-specific tasks. The system unwind library interface accesses the language-specific exception handling semantics via the pointer to the personality routine. The personality routine does not have an ABI-specific name. The size of the personality routine pointer is specified by the pointer encoding used.

'R'

A 'R' may be present at any position after the first character of the string. This character may only be present if 'z' is the first character of the string. If present, The Augmentation Data shall include a 1 byte argument that represents the pointer encoding for the address pointers used in the FDE.

11.6.1.2 The Frame Description Entry Format

Table 11-10 Frame Description Entry Format

Length	Required
Extended Length	Optional
CIE Pointer	Required
PC Begin	Required
PC Range	Required
Augmentation Data Length	Optional
Augmentation Data	Optional
Call Frame Instructions	Required
Padding	

Length

A 4 byte unsigned value indicating the length in bytes of the CIE structure, not including the *Length* field itself. If *Length* contains the value 0xffffffff, then the length is contained the *Extended Length* field. If *Length* contains the value 0, then this CIE shall be considered a terminator and processing shall end.

Extended Length

A 8 byte unsigned value indicating the length in bytes of the CIE structure, not including the *Length* field itself.

CIE Pointer

A 4 byte unsigned value that when subtracted from the offset of the current FDE yields the offset of the start of the associated CIE. This value shall never be 0.

PC Begin

An encoded value that indicates the address of the initial location associated with this FDE. The encoding format is specified in the Augmentation Data.

PC Range

An absolute value that indicates the number of bytes of instructions associated with this FDE.

Augmentation Length

An unsigned LEB128 encoded value indicating the length in bytes of the Augmentation Data. This field is only present if the Augmentation String in the associated CIE contains the character 'z'.

Augmentation Data

A block of data whose contents are defined by the contents of the Augmentation String in the associated CIE as described above. This field is only present if the Augmentation String in the associated CIE contains the character 'z'. The size of this data is given by the Augmentation Length.

Call Frame Instructions

A set of Call Frame Instructions.

Padding

Extra bytes to align the FDE structure to an addressing unit size boundary.

11.6.2 The .eh_frame_hdr section

The `.eh_frame_hdr` section contains additional information about the `.eh_frame` section. A pointer to the start of the `.eh_frame` data, and optionally, a binary search table of pointers to the `.eh_frame` records are found in this section.

Data in this section is encoded according to Section 11.5.1.

Table 11-11 .eh_frame_hdr Section Format

Encoding	Field
unsigned byte	version

Encoding	Field
unsigned byte	eh_frame_ptr_enc
unsigned byte	fde_count_enc
unsigned byte	table_enc
encoded	eh_frame_ptr
encoded	fde_count
	binary search table

version

Version of the .eh_frame_hdr format. This value shall be 1.

eh_frame_ptr_enc

The encoding format of the eh_frame_ptr field.

fde_count_enc

The encoding format of the fde_count field. A value of DW_EH_PE_omit indicates the binary search table is not present.

table_enc

The encoding format of the entries in the binary search table. A value of DW_EH_PE_omit indicates the binary search table is not present.

eh_frame_ptr

The encoded value of the pointer to the start of the .eh_frame section.

fde_count

The encoded value of the count of entries in the binary search table.

binary search table

A binary search table containing fde_count entries. Each entry of the table consist of two encoded values, the initial location, and the address. The entries are sorted in an increasing order by the initial location value.

11.7 Symbol Versioning

11.7.1 Introduction

This chapter describes the Symbol Versioning mechanism. All ELF objects may provide or depend on versioned symbols. Symbol Versioning is implemented by 3 section types: SHT_GNU_verSYM, SHT_GNU_verDEF, and SHT_GNU_verNEED.

The prefix Elf_{xx} in the following descriptions and code fragments stands for either "Elf32" or "Elf64", depending on the architecture.

Versions are described by strings. The structures that are used for symbol versions also contain a member that holds the ELF hashing values of the strings. This allows for more efficient processing.

11.7.2 Symbol Version Table

The special section `.gnu.version` which has a section type of `SHT_GNU_versym` shall contain the Symbol Version Table. This section shall have the same number of entries as the Dynamic Symbol Table in the `.dynsym` section.

The `.gnu.version` section shall contain an array of elements of type `Elfxx_Half`. Each entry specifies the version defined for or required by the corresponding symbol in the Dynamic Symbol Table.

The values in the Symbol Version Table are specific to the object in which they are located. These values are identifiers that are provided by the `vna_other` member of the `Elfxx_Vernaux` structure or the `vd_ndx` member of the `Elfxx_Verdef` structure.

The values 0 and 1 are reserved.

0

The symbol is local, not available outside the object.

1

The symbol is defined in this object and is globally available.

All other values are used to identify version strings located in one of the other Symbol Version sections. The value itself is not the version associated with the symbol. The string identified by the value defines the version of the symbol.

11.7.3 Version Definitions

The special section `.gnu.version_d` which has a section type of `SHT_GNU_verdef` shall contain symbol version definitions. The number of entries in this section shall be contained in the `DT_VERDEFNUM` entry of the Dynamic Section `.dynamic`. The `sh_link` member of the section header (see figure 4-8 in the [System V ABI](#)) shall point to the section that contains the strings referenced by this section.

The section shall contain an array of `Elfxx_Verdef` structures, as described in Figure 11-1, optionally followed by an array of `Elfxx_Verdaux` structures, as defined in Figure 11-2.

```
typedef struct {
    Elfxx_Half  vd_version;
    Elfxx_Half  vd_flags;
    Elfxx_Half  vd_ndx;
    Elfxx_Half  vd_cnt;
    Elfxx_Word  vd_hash;
    Elfxx_Word  vd_aux;
    Elfxx_Word  vd_next;
} Elfxx_Verdef;
```

Figure 11-1 Version Definition Entries

vd_version

Version revision. This field shall be set to 1.

vd_flags

Version information flag bitmask.

vd_ndx

Version index numeric value referencing the `SHT_GNU_versym` section.

vd_cnt

Number of associated verdaux array entries.

vd_hash

Version name hash value (ELF hash function).

vd_aux

Offset in bytes to a corresponding entry in an array of Elfxx_Verdaux structures as defined in Figure 11-2

vd_next

Offset to the next verdef entry, in bytes.

```
typedef struct {
    Elfxx_Word    vda_name;
    Elfxx_Word    vda_next;
} Elfxx_Verdaux;
```

Figure 11-2 Version Definition Auxiliary Entries

vda_name

Offset to the version or dependency name string in the section header, in bytes.

vda_next

Offset to the next verdaux entry, in bytes.

11.7.4 Version Requirements

The special section `.gnu.version_r` which has a section type of `SHT_GNU_verneed` shall contain required symbol version definitions. The number of entries in this section shall be contained in the `DT_VERNEEDNUM` entry of the Dynamic Section `.dynamic`. The `sh_link` member of the section header (see figure 4-8 in [System V ABI](#)) shall point to the section that contains the strings referenced by this section.

The section shall contain an array of Elfxx_Verneed structures, as described in Figure 11-3, optionally followed by an array of Elfxx_Vernaux structures, as defined in Figure 11-4.

```
typedef struct {
    Elfxx_Half    vn_version;
    Elfxx_Half    vn_cnt;
    Elfxx_Word    vn_file;
    Elfxx_Word    vn_aux;
    Elfxx_Word    vn_next;
} Elfxx_Verneed;
```

Figure 11-3 Version Needed Entries

vn_version

Version of structure. This value is currently set to 1, and will be reset if the versioning implementation is incompatibly altered.

vn_cnt

Number of associated verneed array entries.

vn_file

Offset to the file name string in the section header, in bytes.

vn_aux

Offset to a corresponding entry in the vernaux array, in bytes.

vn_next

Offset to the next verneed entry, in bytes.

```
typedef struct {
    Elfxx_Word    vna_hash;
    Elfxx_Half    vna_flags;
    Elfxx_Half    vna_other;
    Elfxx_Word    vna_name;
    Elfxx_Word    vna_next;
} Elfxx_Vernaux;
```

Figure 11-4 Version Needed Auxiliary Entries

vna_hash

Dependency name hash value (ELF hash function).

vna_flags

Dependency information flag bitmask.

vna_other

Object file version identifier used in the .gnu.version symbol version array. Bit number 15 controls whether or not the object is hidden; if this bit is set, the object cannot be used and the static linker will ignore the symbol's presence in the object.

vna_name

Offset to the dependency name string in the section header, in bytes.

vna_next

Offset to the next vernaux entry, in bytes.

11.7.5 Startup Sequence

When loading a sharable object the system shall analyze version definition data from the loaded object to assure that it meets the version requirements of the calling object. This step is referred to as definition testing. The dynamic loader shall retrieve the entries in the caller's `Elfxx_Verneed` array and attempt to find matching definition information in the loaded `Elfxx_Verdef` table.

Each object and dependency shall be tested in turn. If a symbol definition is missing and the `vna_flags` bit for `VER_FLG_WEAK` is not set, the loader shall return an error and exit. If the `vna_flags` bit for `VER_FLG_WEAK` is set in the `Elfxx_Vernaux` entry, and the loader shall issue a warning and continue operation.

When the versions referenced by undefined symbols in the loaded object are found, version availability is certified. The test completes without error and the object shall be made available.