System V Application Binary Interface

Linux Extensions

Version 0.1

Edited by
H.J. Lu[1]

December 30, 2020

[1]hongjiu.lu@intel.com

Contents

List of Tables

List of Figures

Revision History

0.1 — 2016-02-08 Initial draft.

Chapter 1

About this Document

This document contains extensions to to generic System V Application Binary Interface (gABI) available at http://www.sco.com/developers/gabi/latest/contents.html, for Linux.

This document describes the conventions and constraints on the implementation of these extensions for interoperability between various tools.

1.1 Related Information

Links to useful documents:

- Generic System V Application Binary Interface: http://www.sco.com/developers/gabi/latest/contents.html
- Itanium C++ ABI, Revised March 20, 2001: http://mentorembedded.github.io/cxx-abi/

Chapter 2

Object Files

2.1 Sections

The following section types are defined.

Table 2.1: Section types

Name	Value
SHT_GNU_INCREMENTAL_INPUTS	0x6fff4700
SHT_LLVM_ODRTAB	0x6fff4c00
SHT_GNU_ATTRIBUTES	0x6ffffff5
SHT_GNU_HASH	0x6ffffff6
SHT_GNU_LIBLIST	0x6ffffff7

SHT_GNU_INCREMENTAL_INPUTS Incremental build data.

SHT_LLVM_ODRTAB LLVM ODR table.

SHT_GNU_ATTRIBUTES Object attributes.

SHT_GNU_HASH GNU style symbol hash table.

SHT_GNU_LIBLIST List of prelink dependencies.

The section type range 0x6fff4c00 to 0x6fff4cff is reserved for LLVM.

2.1.1 Special Sections

Table 2.2: Special sections

Name	Type	Attributes
.eh_frame	SHT_PROGBITS	SHF_ALLOC
.eh_frame_hdr	SHT_PROGBITS	SHF_ALLOC
.note.ABI-tag	SHT_NOTE	SHF_ALLOC
.note.gnu.build-id	SHT_NOTE	SHF_ALLOC
.note.gnu.property	SHT_NOTE	SHF_ALLOC
.sdata	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE
.sbss	SHT_NOBITS	SHF_ALLOC+SHF_WRITE
.lrodata	SHT_PROGBITS	SHF_ALLOC
.ldata	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE
.lbss	SHT_NOBITS	SHF_ALLOC+SHF_WRITE
.data.rel.ro	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE
.data.rel.local.ro	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE

.eh_frame This section holds the unwind function table. The contents are described in Section 2.1.2 of this document.

.eh_frame_hdr This section holds information about `.eh_frame` section. The contents are described in Section 2.1.3 of this document.

.note.ABI-tag This section holds an ABI note. The contents are described in Section 2.1.6 of this document.

.note.gnu.build-id This section holds a build ID note. The contents are described in Section 2.1.4 of this document.

.note.gnu.property This section holds a program property note. The contents are described in Section 2.1.5 of this document.

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

.sbss This section holds small uninitialized data that contribute to the program's memory image. By definition, the system initializes the data with zeros when the program begins to run.

.lrodata This section holds large read-only data that typically contribute to a non-writable segment in the process image.

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

.lbss This section holds large uninitialized data that contribute to the program's memory image. By definition, the system initializes the data with zeros when the program begins to run.

.data.rel.ro This section holds read-only data that typically contribute to a writable segment in the process image which becomes non-writable after relocation is completed.

.data.rel.local.ro This section holds read-only data that typically contribute to a writable segment in the process image which becomes non-writable after relocation is completed. All relocations contained in this section must be to local objects.

2.1.2 EH_FRAME section

The call frame information needed for unwinding the stack is output into one section named `.eh_frame`. An `.eh_frame` section consists of one or more subsections. Each subsection contains a CIE (Common Information Entry) followed by varying number of FDEs (Frame Descriptor Entry). A FDE corresponds to an explicit or compiler generated function in a compilation unit, all FDEs can access the CIE that begins their subsection for data. If the code for a function is not one contiguous block, there will be a separate FDE for each contiguous sub-piece.

If an object file contains C++ template instantiations there shall be a separate CIE immediately preceding each FDE corresponding to an instantiation.

Using the preferred encoding specified below, the `.eh_frame` section can be entirely resolved at link time and thus can become part of the text segment.

`EH_PE` encoding below refers to the pointer encoding as specified in Section `DWARF Exception Header Encoding` of Linux Standard Base Core Specification.

Table 2.3: Common Information Entry (CIE)

Field	Length (byte)	Description
Length	4	Length of the CIE (not including this 4-byte field)
CIE id	4	Value 0 for `.eh_frame` (used to distinguish CIEs and FDEs when scanning the section)
Version	1	Value One (1)
CIE Augmentation String	string	Null-terminated string with legal values being "" or 'z' optionally followed by single occurrances of 'P', 'L', or 'R' in any order. The presence of character(s) in the string dictates the content of field 8, the Augmentation Section. Each character has one or two associated operands in the AS (see table 2.4 for which ones). Operand order depends on position in the string ('z' must be first).
Code Align Factor	uleb128	To be multiplied with the "Advance Location" instructions in the Call Frame Instructions
Data Align Factor	sleb128	To be multiplied with all offsets in the Call Frame Instructions
Ret Address Reg	1/uleb128	A "virtual" register representation of the return address. In Dwarf V2, this is a byte, otherwise it is uleb128. It is a byte in gcc 3.3.x
Optional CIE Augmentation Section	varying	Present if Augmentation String in Augmentation Section field 4 is not 0. See table 2.4 for the content.
Optional Call Frame Instructions	varying	

Table 2.4: CIE Augmentation Section Content

Char	Operands	Length (byte)	Description
z	size	uleb128	Length of the remainder of the Augmentation Section
P	personality_enc	1	Encoding specifier - preferred value is a pc-relative, signed 4-byte
	personality routine	(encoded)	Encoded pointer to personality routine (actually to the PLT entry for the personality routine)
R	code_enc	1	Non-default encoding for the code-pointers (FDE members `initial_location` and `address_range` and the operand for `DW_CFA_set_loc`) - preferred value is pc-relative, signed 4-byte
L	lsda_enc	1	FDE augmentation bodies may contain LSDA pointers. If so they are encoded as specified here - preferred value is pc-relative, signed 4-byte possibly indirect thru a GOT entry

Table 2.5: Frame Descriptor Entry (FDE)

Field	Length (byte)	Description
Length	4	Length of the FDE (not including this 4-byte field)
CIE pointer	4	Distance from this field to the nearest preceding CIE (the value is subtracted from the current address). This value can never be zero and thus can be used to distinguish CIE's and FDE's when scanning the `.eh_frame` section
Initial Location	var	Reference to the function code corresponding to this FDE. If 'R' is missing from the CIE Augmentation String, the field is an 8-byte absolute pointer. Otherwise, the corresponding `EH_PE` encoding in the CIE Augmentation Section is used to interpret the reference
Address Range	var	Size of the function code corresponding to this FDE. If 'R' is missing from the CIE Augmentation String, the field is an 8-byte unsigned number. Otherwise, the size is determined by the corresponding `EH_PE` encoding in the CIE Augmentation Section (the value is always absolute)
Optional FDE Augmentation Section	var	Present if CIE Augmentation String is non-empty. See table 2.6 for the content.
Optional Call Frame Instructions	var	

Table 2.6: FDE Augmentation Section Content

Char	Operands	Length (byte)	Description
z	length	uleb128	Length of the remainder of the Augmentation Section
L	LSDA	var	LSDA pointer, encoded in the format specified by the corresponding operand in the CIE's augmentation body. (only present if length > 0).

The existence and size of the optional call frame instruction area must be computed based on the overall size and the offset reached while scanning the preceding fields of the CIE or FDE.

The overall size of a `.eh_frame` section is given in the ELF section header. The only way to determine the number of entries is to scan the section until the end, counting entries as they are encountered.

2.1.3 EH_FRAME_HDR section

`.eh_frame_hdr` section contains information about `.eh_frame` section for optimizing stack unwinding.

Table 2.7: `.eh_frame_hdr` Section Format

Encoding	Field	Required
unsigned byte	version	Yes
unsigned byte	eh_frame_ptr_enc	Yes
unsigned byte	fde_count_enc	Yes
unsigned byte	table_enc	Yes
[encoded]	eh_frame_ptr	No
[encoded]	fde_count	No
	binary search table	No

version Version of `.eh_frame_hdr` section format. It should be 1.

eh_frame_ptr_enc `EH_PE` encoding of pointer to start of `.eh_frame` section.

fde_count_enc `EH_PE` encoding of total FDE count number. `DW_EH_PE_omit` if there is no `binary search table`.

table_enc `EH_PE` encoding of `binary search table`. `DW_EH_PE_omit` if there is no `binary search table`.

eh_frame_ptr Pointer to start of `.eh_frame` section.

fde_count Total number of FDEs in `.eh_frame` section.

binary search table A binary search table containing `fde_count` entries. Each entry consists of FDE initial location and address. The entries are sorted in the increasing order by FDE initial location value.

2.1.4 `.note.gnu.build-id` section

`.note.gnu.build-id` section contains a build ID note which is unique among the set of meaningful contents for ELF files and identical when the output file would otherwise have been identical. It can be merged with other `SHT_NOTE` sections.

Table 2.8: The Build ID Note Format

Field	Length	Contents
`n_namsz`	`4`	`4`
`n_descsz`	`4`	The note descriptor size
`n_type`	`4`	`NT_GNU_BUILD_ID`
`n_name`	`4`	`GNU`
`n_desc`	`n_descsz`	The build ID

n_namsz Size of the `n_name` field. A 4-byte integer in the format of the target processor. It should be 4.

n_descsz Size of the `n_desc` field. A 4-byte integer in the format of the target processor.

n_type Type of the note descriptor. A 4-byte integer in the format of the target processor. It should be `NT_GNU_BUILD_ID`.

n_name Owner of the build ID note. A null-terminated character string. It should be `GNU`.

n_desc The note descriptor. The first `n_descsz` bytes in `n_desc` is the build ID.

2.1.5 `.note.gnu.property` section

`.note.gnu.property` section contains a program property note which describes special handling requirements for linker and run-time loader. It can be merged with other `SHT_NOTE` sections.

Table 2.9: The Program Property Note Format

Field	Length	Contents
`n_namsz`	4	4
`n_descsz`	4	The note descriptor size
`n_type`	4	`NT_GNU_PROPERTY_TYPE_0`
`n_name`	4	`GNU`
`n_desc`	`n_descsz`	The program property array

n_namsz Size of the `n_name` field. A 4-byte integer in the format of the target processor. It should be 4.

n_descsz Size of the `n_desc` field. A 4-byte integer in the format of the target processor.

n_type Type of the note descriptor. A 4-byte integer in the format of the target processor. It should be `NT_GNU_PROPERTY_TYPE_0`.

n_name Owner of the program property note. A null-terminated character string. It should be `GNU`.

n_desc The note descriptor. The first `n_descsz` bytes in `n_desc` is the program property array.

The program property array

Each array element represents one program property with type, data size and data. In 64-bit objects, each element is an array of 8-byte integers in the format of the target processor. In 32-bit objects, each element is an array of 4-byte integers in the format of the target processor. An array element has the following structure:

```
typedef struct {
  Elf_Word pr_type;
  Elf_Word pr_datasz;
  unsigned char pr_data[PR_DATASZ];
  unsigned char pr_padding[PR_PADDING];
} Elf_Prop;
```

pr_type The type of program property. A 4-byte integer in the format of the target processor.

pr_datasz The size of the `pr_data` field. A 4-byte integer in the format of the target processor.

pr_data The program property descriptor which is aligned to 4 bytes in 32-bit objects and 8 bytes in 64-bit objects.

pr_padding The padding. If necessary, it aligns the array element to 8 or 4-byte alignment (depending on whether the file is a 64-bit or 32-bit object).

PR_DATASZ The value in the `pr_datasz` field. A constant.

PR_PADDING The size of the `pr_padding` field. A constant.

The array elements are sorted by the program property type in ascending order.

Types of program properties

The following program property types are defined:

Table 2.10: Program Property Types

Name	Value
GNU_PROPERTY_STACK_SIZE	1
GNU_PROPERTY_NO_COPY_ON_PROTECTED	2
GNU_PROPERTY_LOPROC	0xc0000000
GNU_PROPERTY_HIPROC	0xdfffffff
GNU_PROPERTY_LOUSER	0xe0000000
GNU_PROPERTY_HIUSER	0xffffffff

GNU_PROPERTY_STACK_SIZE Its pr_data field contains a 4-byte integer in 32-bit objects and 8-byte integer in 64-bit objects, in the format of the target processor. Linker should select the maximum value among all input relocatable objects and copy this property to the output. Run-time loader should raise the stack limit to the value specified in this property.

GNU_PROPERTY_NO_COPY_ON_PROTECTED This indicates that there should be no copy relocations against protected data symbols. If a relocatable object contains this property, linker should treat protected data symbol as defined locally at run-time and copy this property to the output share object. Linker should add this property to the output share object if any protected symbol is expected to be defined locally at run-time. Run-time loader should disallow copy relocations against protected data symbols defined in share objects with GNU_PROPERTY_NO_COPY_ON_PROTECTED property. Its PR_DATASZ should be 0.

GNU_PROPERTY_LOPROC through GNU_PROPERTY_HIPROC Values in this inclusive range are reserved for processor-specific semantics.

GNU_PROPERTY_LOUSER through GNU_PROPERTY_HIUSER Values in this inclusive range are reserved for application-specific semantics.

2.1.6 .note.ABI-tag section

.note.ABI-tag section contains an ABI note which is used to identify OS and version targeted. It can be merged with other SHT_NOTE sections.

Table 2.11: The ABI Tag Note Format

Field	Length	Contents
n_namsz	4	4
n_descsz	4	16
n_type	4	NT_GNU_ABI_TAG
n_name	4	GNU
n_desc	16	The ABI tag

n_namsz Size of the n_name field. A 4-byte integer in the format of the target processor. It should be 4.

descsz Size of the n_desc field. It should be 16.

n_type Type of the note descriptor. A 4-byte integer in the format of the target processor. It should be NT_GNU_ABI_TAG.

n_name Owner of the build ID note. A null-terminated character string. It should be GNU.

n_desc The note descriptor. Four 4-byte integers in the format of the target processor. The first 4-byte integer should 0. The second, third, and fourth 4-byte integers contain the earliest compatible kernel version. For example, if the 3 integers are 2, 2, and 5, this signifies a 2.2.5 kernel.

2.1.7 Alignment of Note Sections

All entries in a PT_NOTE segment have the same alignment which equals to the p_align field in program header.

According to gABI, each note entry should be aligned to 4 bytes in 32-bit objects or 8 bytes in 64-bit objects. But .note.ABI-tag section (see Section 2.1.6) and .note.gnu.build-id section (see Section 2.1.4) are aligned to 4 bytes in both 32-bit and 64-bit objects. Note parser should use p_align for note alignment, instead of assuming alignment based on ELF file class.

2.2 Symbol Table

Table 2.12: Linux Specific Symbol Types

Name	Value
STT_GNU_IFUNC	10

2.2.1 STT_GNU_IFUNC Symbol

This symbol type is the same as STT_FUNC except that it always points to a resolve function or piece of executable code which takes no arguments and returns a function pointer. If an STT_GNU_IFUNC symbol is referred to by a relocation, then evaluation of that relocation is delayed until load-time. The value used in the relocation is the function pointer returned by an invocation of the STT_GNU_IFUNC symbol.

The purpose of the STT_GNU_IFUNC symbol type is to allow the run-time to select between multiple versions of the implementation of a specific function. The selection made in general will take the currently available hardware into account and select the most appropriate version.

Implementation Considerations

The calling convention of the STT_GNU_IFUNC resolve function, which takes no arguments and returns a function pointer, should follow the processor-specific ABI. All rules for caller-saved and callee-saved registers apply.

There are special considerations for GOT when PLT is required:

- All references to a STT_GNU_IFUNC symbol, including function call and function pointer, should go through the PLT slot, which jumps to the address stored in the GOT entry. If the STT_GNU_IFUNC symbol is locally defined, a processor-specific IRELATIVE relocation should be applied to the GOT entry at load time. Otherwise, dynamic linker will lookup the symbol at the first reference to the function and update the GOT entry. This applies to all usages of STT_GNU_IFUNC symbols in shared library, dynamic executable and static executable.

Instead of branching to an `STT_GNU_IFUNC` symbol directly, calling a function always branches to its PLT entry, which simply loads its GOTPLT entry and branches to it. Its GOTPLT entry has the real function address.

- An `STT_GNU_IFUNC` symbol has an optional GOT entry for the function pointer value of the symbol. To load an `STT_GNU_IFUNC` symbol function pointer value:
  - Use its GOTPLT entry in a shared object if it is forced local or not dynamic.
  - Use its GOTPLT entry in a non-shared object if pointer equality isn't needed.
  - Use its GOTPLT entry in a position independent executable (PIE).
  - Use its GOTPLT entry if no normal GOT, other than GOTPLT, is used.
  - Otherwise use its GOT entry. We only need to relocate its GOT entry in a shared object.
- We need dynamic relocation for `STT_GNU_IFUNC` symbol only when there is a non-GOT reference in a shared object.
- When a shared library references a `STT_GNU_IFUNC` symbol defined in executable, the address of the resolved function may be used. But in non-shared executable, the address of its GOTPLT entry may be used. Pointer equality may not work correctly. PIE should be used if pointer equality is required.

Chapter 3

Program Loading and Dynamic Linking

3.1 Program header

The following Linux program header types are defined:

Table 3.1: Program Header Types

Name	Value
PT_GNU_EH_FRAME	0x6474e550
PT_GNU_PROPERTY	0x6474e553

PT_GNU_EH_FRAME The segment contains `.eh_frame_hdr` section. See Section 2.1.3 of this document.

PT_GNU_PROPERTY The segment contains `.note.gnu.property` section. See Section 2.1.5 of this document.

3.2 Note Section

The following note descriptor types are defined:

Table 3.2: Note Descriptor Types

Name	Value
NT_GNU_ABI_TAG	1
NT_GNU_BUILD_ID	3
NT_GNU_PROPERTY_TYPE_0	5

NT_GNU_ABI_TAG The ABI tag note. See Section 2.1.6 of this document.

NT_GNU_BUILD_ID The build ID note. See Section 2.1.4 of this document.

NT_GNU_PROPERTY_TYPE_0 The program property note. See Section 2.1.5 of this document.

Chapter 4

Development Environment

During compilation of C or C++ code at least the symbols in table 4.1 are defined by the pre-processor.

Table 4.1: Predefined Pre-Processor Symbols

`__linux`
`__linux__`
`__unix`
`__unix__`