The SFrame Format

Version 3

***** DRAFT - NOT FOR DISTRIBUTION *****

26 July 2026

Indu Bhagat

Table of Contents

1 Introduction

1.1 Overview

The SFrame stack trace information is provided in a loaded section, named `.sframe`. When available, the `.sframe` section appears in segment of type `PT_GNU_SFRAME`. An ELF SFrame section will have the type `SHT_GNU_SFRAME`.

The SFrame format is currently supported only for select ABIs, namely, AMD64, AAPCS64, and s390x.

A portion of the SFrame format follows an unaligned on-disk representation. Some data structures, however, (namely the SFrame header and the SFrame function descriptor index) have elements at their natural boundaries. All data structures are packed, unless otherwise stated.

The contents of the SFrame section are stored in the target endianness, i.e., in the endianness of the system on which the section is targeted to be used. An SFrame section reader may use the magic number in the SFrame header to identify the endianness of the SFrame section.

Addresses in this specification are expressed in bytes. The use of term 'data word' in this document is colloquial; it should not be understood to correlate with the architectural machine word or any specific hardware data width.

The rest of this specification describes the current version of the format, `SFRAME_VERSION_3`, in detail. Additional sections outline the major changes made to each previously published version of the SFrame stack trace format.

This document is intended to be in sync with the C code in `sframe.h`. Please report discrepancies between the two, if any.

1.2 Changes from Version 2 to Version 3

The following is a list of the changes made to the SFrame stack trace format since Version 2 was published. Note that SFrame Version 2 had up to two Errata.

- Terminology improvements and renames for readability
  - Use the terminology 'PC offset' in place of 'Addr' for function start PC offset consistently.
  - Make a distinction between SFrame FDE Type (e.g., `SFRAME_FDE_TYPE_DEFAULT`, `SFRAME_FDE_TYPE_FLEX`) vs SFrame FDE PC Type (i.e., `SFRAME_FDE_PCTYPE_MASK`, `SFRAME_FDE_PCTYPE_INC`).
  - Instead of using the term 'info word', use a more precise term 'info byte' in specification for the info bytes in SFrame FDE and SFrame FRE.
  - Use term 'data word' instead of 'offset' to convey the functional role of the variable-length array of bytes trailing the SFrame FRE header. With the introduction of flexible FDE type, the interpretation of those bytes is not always as an offset.
  - Rename `SFRAME_FRE_OFFSET_<N>B` to `SFRAME_FRE_DATAWORD_<N>B`.
- Reorganize the SFrame function descriptor entry into two distinct structures:
  - SFrame function descriptor index

  - SFrame function descriptor attribute

  Rename structure members as a consequence.
- Narrow the width of `sfda_func_num_fres` to `uint16_t` and remove padding field `sfde_func_padding2`.
- Increase the width of the `sfdi_func_start_offset` to `int64_t`. This field is renamed from the `sfde_func_start_address` in SFrame Version 2 specification.
- Signal frames are marked with one bit in `sfda_func_info`.
- Addition of a new function info byte `sfda_func_info2` in SFrame function descriptor attribute structure to store additional information about the stack trace data for the function.
- Reserve 5-bits for FDE types. Define two FDE types: default FDE type `SFRAME_FDE_TYPE_DEFAULT`, and flexible FDE type `SFRAME_FDE_TYPE_FLEX`.
- Define a new FDE type `SFRAME_FDE_TYPE_FLEX` to convey stack trace information for specific cases, e.g., when CFA is non-SP/FP based, or when FP/RA recovery is REG-based.
- An SFrame FDE of type `SFRAME_FDE_TYPE_DEFAULT` with no FREs is used to indicate an outermost frame.
- On s390x, use FDE type `SFRAME_FDE_TYPE_FLEX` to encode FP/RA recovery from REG, instead of encoding DWARF register number in the SFrame FRE variable-length data of FDE type `SFRAME_FDE_TYPE_DEFAULT`.

1.3 Changes from Version 1 to Version 2

The following is a list of the changes made to the SFrame stack trace format since Version 1 was published.

- Add an unsigned 8-bit integral field to the SFrame function descriptor entry to encode the size of the repetitive code blocks. Such code blocks, e.g., pltN entries, use an SFrame function descriptor entry of type `SFRAME_FDE_PCTYPE_MASK`.
- Add an unsigned 16-bit integral field to the SFrame function descriptor entry to serve as padding. This helps ensure natural alignment for the members of the data structure.
- The above two imply that each SFrame function descriptor entry has a fixed size of 20 bytes instead of its size of 17 bytes in SFrame format Version 1.
- [Errata 1] Add a new flag SFRAME_F_FDE_FUNC_START_PCREL, as an erratum to SFrame Version 2, to indicate the encoding of the SFrame FDE function start address field:
  - if set, `sfde_func_start_offset` field contains the offset in bytes to the start PC of the associated function from the field itself.
  - if unset, `sfde_func_start_offset` field contains the offset in bytes to the start PC of the associated function from the start of the SFrame section.
- [Errata 1] Add a new ABI/arch identifier SFRAME_ABI_S390X_ENDIAN_BIG for the s390 architecture (64-bit) s390x ABI. Other s390x-specific backward compatible changes including the following helper definitions have been incrementally added to SFrame Version 2 only:
  - SFRAME_S390X_SP_VAL_OFFSET: SP value offset from CFA.

  - SFRAME_V2_S390X_OFFSET_IS_REGNUM: Test whether FP/RA offset is an encoded DWARF register number.
  - SFRAME_V2_S390X_OFFSET_ENCODE_REGNUM: Encode a DWARF register number as an FP/RA offset.
  - SFRAME_V2_S390X_OFFSET_DECODE_REGNUM: Decode a DWARF register number from an FP/RA offset.
  - SFRAME_FRE_RA_OFFSET_INVALID: Invalid RA offset value (like SFRAME_CFA_FIXED_RA_INVALID). Used on s390x as padding offset to represent FP without RA saved.
  - SFRAME_S390X_CFA_OFFSET_ADJUSTMENT: CFA offset (from CFA base register) adjustment value. Used to enable use of 8-bit SFrame offsets on s390x.
  - SFRAME_S390X_CFA_OFFSET_ALIGNMENT_FACTOR: CFA offset alignment factor. Used to scale down the CFA offset to improve the use of 8-bit SFrame offsets.
  - SFRAME_V2_S390X_CFA_OFFSET_ENCODE: Encode CFA offset (i.e., apply CFA offset adjustment and then scale down by CFA offset alignment factor).
  - SFRAME_V2_S390X_CFA_OFFSET_DECODE: Decode CFA offset (i.e., scale up by CFA offset alignment factor and then revert CFA offset adjustment).
- [Errata 1] An ELF SFrame section has the type `SHT_GNU_SFRAME`.
- [Errata 2] An offset count of zero in the SFrame FRE info byte indicates that the return address (RA) is undefined for the range of PCs covered by the SFrame FRE. A stack tracer may use this as indication that an outermost frame has been reached and the stack trace is complete.

SFrame Version 1 is now obsolete and should not be used.

2 SFrame Section

The SFrame section consists of an SFrame header, starting with a preamble, and two other sub-sections, namely the SFrame function descriptor entry (SFrame FDE) sub-section, and the SFrame frame row entry (SFrame FRE) sub-section.

2.1 SFrame Preamble

The preamble is a 32-bit packed structure; the only part of the SFrame section whose format cannot vary between versions.

```
typedef struct sframe_preamble
{
  uint16_t sfp_magic;
  uint8_t sfp_version;
  uint8_t sfp_flags;
} ATTRIBUTE_PACKED sframe_preamble;
```

Every element of the SFrame preamble is naturally aligned.

All values are stored in the endianness of the target system for which the SFrame section is intended. Further details:

Offset	Type	Name	Description
0x00	`uint16_t`	`sfp_magic`	The magic number for SFrame section: 0xdee2. Defined as a macro `SFRAME_MAGIC`.
0x02	`uint8_t`	`sfp_version`	The version number of this SFrame section. See Section 2.1.2 [SFrame Version], page 4, for the set of valid values. Current version is `SFRAME_VERSION_3`.
0x03	`uint8_t`	`sfp_flags`	Flags (section-wide) for this SFrame section. See Section 2.1.3 [SFrame Flags], page 5, for the set of valid values.

2.1.1 SFrame Magic Number and Endianness

SFrame sections are stored in the target endianness of the system that consumes them. A consumer library reading or writing SFrame sections should detect foreign-endianness by inspecting the SFrame magic number in the `sfp_magic` field in the SFrame header. It may then provide means to endian-flip the SFrame section as necessary.

2.1.2 SFrame Version

The version of the SFrame format can be determined by inspecting `sfp_version`. The following versions are currently valid:

Version Name	Number	Description
`SFRAME_VERSION_1`	1	First version, obsolete.
`SFRAME_VERSION_2`	2	Second version.
`SFRAME_VERSION_3`	3	Third version, under development.

This document describes `SFRAME_VERSION_3`.

2.1.3 SFrame Flags

The preamble contains bitflags in its `sfp_flags` field that describe various section-wide properties.

The following flags are currently defined.

Flag	Version	Value	Meaning
`SFRAME_F_FDE_SORTED`	All	0x1	Function Descriptor Entries are sorted on PC.
`SFRAME_F_FRAME_POINTER`	1-2	0x2	All functions in the object file preserve frame pointer.
`SFRAME_F_FDE_FUNC_START_PCREL`	2-3	0x4	The `sfdi_func_start_offset` field in the SFrame FDE is an offset in bytes to the function's start address, from the field itself. If unset, the `sfdi_func_start_offset` field in the SFrame FDE is an offset in bytes to the function's start address, from the start of the SFrame section.

The purpose of `SFRAME_F_FRAME_POINTER` flag was to facilitate stack tracers to reliably fallback on the frame pointer based stack tracing method, if SFrame information is not present for some function in the SFrame section.

Further flags may be added in future. Bits corresponding to the currently undefined flags must be set to zero.

2.2 SFrame Header

The SFrame header is the first part of an SFrame section. It begins with the SFrame preamble. All parts of it other than the preamble (see Section 2.1 [SFrame Preamble], page 4) can vary between SFrame file versions. It contains metadata that apply to the section as a whole, and offsets to the various other sub-sections defined in the format. As with the rest of the SFrame section, all values are stored in the endianness of the target system.

The two sub-sections tile the SFrame section: each section runs from the offset given until the start of the next section. An explicit length is given for the last sub-section, the SFrame Frame Row Entry (SFrame FRE) sub-section.

```
typedef struct sframe_header
{
  sframe_preamble sfh_preamble;
  uint8_t sfh_abi_arch;
  int8_t sfh_cfa_fixed_fp_offset;
  int8_t sfh_cfa_fixed_ra_offset;
  uint8_t sfh_auxhdr_len;
  uint32_t sfh_num_fdes;
  uint32_t sfh_num_fres;
  uint32_t sfh_fre_len;
  uint32_t sfh_fdeoff;
```

```
  uint32_t sfh_freoff;
} ATTRIBUTE_PACKED sframe_header;
```

Every element of the SFrame header is naturally aligned.

The sub-section offsets, namely `sfh_fdeoff` and `sfh_freoff`, in the SFrame header are relative to the *end* of the SFrame header; they are each an offset in bytes into the SFrame section where the SFrame FDE sub-section and the SFrame FRE sub-section respectively start.

The SFrame section contains `sfh_num_fdes` number of fixed-length array elements in the SFrame FDE sub-section. Each array element is of type SFrame function descriptor entry; each providing a high-level function description for the purpose of stack tracing. More details in Section 2.3 [SFrame Function Descriptor Entries], page 7.

Next, the SFrame FRE sub-section, starting at offset `sfh_fre_off`, describes the stack trace information for each function. For each function, the SFrame FRE sub-section contains the SFrame FDE attribute data and `sfh_num_fres` number of variable-length array elements. Each array element is of type SFrame frame row entry. See Section 2.4 [SFrame Frame Row Entries], page 12.

SFrame header allows specifying explicitly the fixed offsets from CFA, if any, from which FP or RA may be recovered. For example, in AMD64, the stack offset of the return address is `CFA - 8`. Since these offsets are expected to be in close vicinity to the CFA in most ABIs, `sfh_cfa_fixed_fp_offset` and `sfh_cfa_fixed_ra_offset` are limited to signed 8-bit integers.

The SFrame format has made some provisions for supporting more ABIs/architectures in the future. One of them is the concept of the auxiliary SFrame header. Bytes in the auxiliary SFrame header may be used to convey further ABI-specific information. The `sframe_header` structure provides an unsigned 8-bit integral field to denote the size (in bytes) of an auxiliary SFrame header. The auxiliary SFrame header follows right after the `sframe_header` structure. As for the calculation of the sub-section offsets, namely `sfh_fdeoff` and `sfh_freoff`, the *end* of SFrame header must be the end of the auxiliary SFrame header, if the latter is present.

Putting it all together:

Offset	Type	Name	Description
0x00	`sframe_preamble`	`sfh_preamble`	The SFrame preamble. See Section 2.1 [SFrame Preamble], page 4.
0x04	`uint8_t`	`sfh_abi_arch`	The ABI/arch identifier. See Section 2.2.1 [SFrame ABI/arch Identifier], page 7.
0x05	`int8_t`	`sfh_cfa_fixed_fp_offset`	The CFA fixed FP offset, if any.
0x06	`int8_t`	`sfh_cfa_fixed_ra_offset`	The CFA fixed RA offset, if any.

0x07	`uint8_t`	`sfh_auxhdr_len`	Size in bytes of the auxiliary header that follows the `sframe_header` structure.
0x08	`uint32_t`	`sfh_num_fdes`	The number of SFrame FDEs in the section.
0x0c	`uint32_t`	`sfh_num_fres`	The number of SFrame FREs in the section.
0x10	`uint32_t`	`sfh_fre_len`	The length in bytes of the SFrame FRE sub-section.
0x14	`uint32_t`	`sfh_fdeoff`	The offset in bytes to the SFrame FDE sub-section.
0x18	`uint32_t`	`sfh_freoff`	The offset in bytes to the SFrame FRE sub-section.

2.2.1 SFrame ABI/arch Identifier

SFrame header identifies the ABI/arch of the target system for which the executable and hence, the stack trace information contained in the SFrame section, is intended. There are currently four identifiable ABI/arch values in the format.

ABI/arch Identifier	**Value**	**Description**
`SFRAME_ABI_AARCH64_ENDIAN_BIG`	1	AArch64 big-endian
`SFRAME_ABI_AARCH64_ENDIAN_LITTLE`	2	AArch64 little-endian
`SFRAME_ABI_AMD64_ENDIAN_LITTLE`	3	AMD64 little-endian
`SFRAME_ABI_S390X_ENDIAN_BIG`	4	s390x big-endian

The presence of an explicit identification of ABI/arch in SFrame may allow stack trace generators to make certain ABI/arch-specific decisions.

2.3 SFrame FDE

SFrame function descriptor entry is a conceptual entity which contains the function-level metadata necessary for stack tracing through the function. It is composed of two physical entities: the SFrame function descriptor index (SFrame FDE index) and the SFrame function descriptor attribute (SFrame FDE attribute). Both SFrame FDE index and SFrame FDE attribute are fixed-length structures, albeit with different alignment guarantees.

2.3.1 The SFrame FDE Index

The SFrame FDE index entries are stored in a sub-section of their own, forming a searchable index. If the SFrame header flag `SFRAME_F_FDE_SORTED` is set, then the entries are sorted by `sfdi_func_start_offset`, allowing for efficient binary search. Typically (as is the case with GNU ld) a linked object or executable will have the `SFRAME_F_FDE_SORTED` set. This makes the job of a stack tracer easier as it may then employ a binary search scheme to look for the stack trace information pertinent to a given PC.

```
typedef struct sframe_func_desc_idx
{
  int64_t sfdi_func_start_offset;
  uint32_t sfdi_func_size;
  uint32_t sfdi_func_start_fre_off;
} ATTRIBUTE_PACKED sframe_func_desc_idx;
```

Each entry of the SFrame function descriptor index is naturally aligned. The following table describes each component of the SFrame FDE index entry:

Offset	Type	Name	Description
0x00	`int64_t`	`sfdi_func_start_offset`	Signed 64-bit integral field specifying the offset to the start address of the described function. If the flag `SFRAME_F_FDE_FUNC_START_PCREL`, See Section 2.1.3 [SFrame Flags], page 5, in the SFrame header is set, the value encoded in the `sfdi_func_start_offset` field is the offset in bytes to the function's start address from the `sfdi_func_start_offset` field itself. Otherwise, it is the offset in bytes from the start of the SFrame section.
0x08	`uint32_t`	`sfdi_func_size`	Unsigned 32-bit integral field specifying the size of the function in bytes.
0x0c	`uint32_t`	`sfdi_func_start_fre_off`	Unsigned 32-bit integral field specifying the offset to the start of the function's stack trace data (SFrame FREs). This offset is relative to the *beginning of the SFrame FRE sub-section.*

2.3.2 The SFrame FDE Attribute

The SFrame FDE attribute structure provides information about the SFrame FRE entries that follow: their number and their encoding. The SFrame FDE attributes are stored at the beginning of each function's stack trace data within the SFrame FRE sub-section. Because these structures are interleaved with variable-length FREs, their elements are not guaranteed to be at naturally aligned boundaries.

```
typedef struct sframe_func_desc_attr
```

```
{
  uint16_t sfda_func_num_fres;
  uint8_t sfda_func_info;
  uint8_t sfda_func_info2;
  uint8_t sfda_func_rep_size;
} ATTRIBUTE_PACKED sframe_func_desc_attr;
```

Following table describes each component of the SFrame FDE attribute:

Offset	Type	Name	Description
0x00	`uint16_t`	`sfda_func_num_fres`	Unsigned 16-bit integral field specifying the total number of SFrame FREs used for the function.
0x02	`uint8_t`	`sfda_func_info`	Unsigned 8-bit integral field specifying the SFrame FDE info byte.
0x03	`uint8_t`	`sfda_func_info2`	Additional unsigned 8-bit integral field specifying the SFrame FDE info byte.
0x04	`uint8_t`	`sfda_func_rep_size`	Unsigned 8-bit integral field specifying the size of the repetitive code block for which an SFrame FDE of type `SFRAME_FDE_PCTYPE_MASK` is used. For example, in AMD64, the size of a pltN entry is 16 bytes.

`sfda_func_info` and `sfda_func_info2` are the SFrame FDE **Info Bytes**, containing information like the FRE type and their encoding, and the FDE type for the function. See Section 2.3.3 [The SFrame FDE Info Bytes], page 9.

The SFrame FDE attribute has some currently unused bits in the SFrame FDE info bytes, that may be used for the purpose of extending the SFrame file format specification for future ABIs. See Section 2.3.3.2 [The SFrame FDE Types], page 11, subsection.

2.3.3 The SFrame FDE Info Bytes

The SFrame FDE Attribute contains two distinct bytes, `sfda_func_info` and `sfda_func_info2`. Together these are referred to as the SFrame FDE info bytes. These bytes contain vital information necessary to:

- read and interpret SFrame FRE data, e.g., the number and size of each SFrame FRE offset,
- PC Type for SFrame FDE,
- type of SFrame FDE,
- size of repeat block, if PC Type is `SFRAME_FDE_PCTYPE_MASK`.

The first info byte `sfda_func_info` is a bitfield split into four parts. From MSB to LSB:

Bit offset	Name	Description
7	`signal_p`	Signal frame.

6	`unused`	Unused bit.
5	`pauth_key`	(For AArch64) Specify which key is used for signing the return addresses in the SFrame FDE. Two possible values: `SFRAME_AARCH64_PAUTH_KEY_A` (0), or `SFRAME_AARCH64_PAUTH_KEY_B` (1). Unsed in AMD64, s390x
4	`fde_pctype`	Specify the SFrame FDE PC Type. Two possible values: `SFRAME_FDE_PCTYPE_MASK` (1), or `SFRAME_FDE_PCTYPE_INC` (0). See Section 2.3.3.1 [The SFrame FDE PC Types], page 10.
0–3	`fre_type`	Choice of three SFrame FRE types. See Section 2.3.3.3 [The SFrame FRE Types], page 11.

The second info byte `sfda_func_info2` is a bitfield split into two parts. From MSB to LSB:

Bit offset	**Name**	**Description**
7–5	`unused`	Unused bits.
4–0	`fde_type`	Specify the SFrame FDE type. Two possible values: `SFRAME_FDE_TYPE_DEFAULT` (0), or `SFRAME_FDE_TYPE_FLEX` (1). See Section 2.3.3.2 [The SFrame FDE Types], page 11.

2.3.3.1 The SFrame FDE PC Types

The SFrame format defines two types of FDE PC types. The choice of which SFrame FDE PC type to use is made based on the instruction patterns in the relevant program stub.

An FDE of PC type `SFRAME_V3_FDE_PCTYPE_INC` contains FREs whose PCs are to be interpreted as the address of a single instruction, measured in bytes and relative to the beginning of the function.

In contrast, a FDE of PC type `SFRAME_V3_FDE_PCTYPE_MASK` contains FREs whose PCs are to be interpreted as masks that identify several instructions. This is useful for cases where a small pattern of instructions in a program stub is used repeteadly for a specific functionality, like PLT entries and trampolines.

Name of SFrame FDE PC Type	**Value**	**Description**
`SFRAME_V3_FDE_PCTYPE_INC`	0	Stacktracers perform a (PC >= FRE_START_ADDR) to look up a matching FRE.

`SFRAME_V3_FDE_PCTYPE_MASK`	1	Stacktracers perform a (PC % REP_BLOCK_SIZE >= FRE_START_ADDR) to look up a matching FRE. REP_BLOCK_SIZE is the size in bytes of the repeating block of program instructions and is encoded via `sfde_func_rep_size` in the SFrame FDE.

2.3.3.2 The SFrame FDE Types

The SFrame format defines two types of Function Descriptor Entries (FDEs) to encode stack trace information. The choice of FDE type determines how the data in the variable-length Frame Row Entries (FREs) is interpreted. The FDE type is encoded in the lower 5 bits of the `sfda_func_info2` field in the SFrame FDE attribute.

Name	Value	Description
`SFRAME_FDE_TYPE_DEFAULT`	0	The default FDE type. CFA is recovered using the Stack Pointer (SP) or Frame Pointer (FP) plus a signed offset. Return Address (RA) and Frame Pointer (FP) are recovered using the CFA plus a signed offset (or a fixed register for specific architectures like s390x). The variable-length array of bytes trailing each SFrame FRE are interpreted according to the ABI/arch-specific rules for the target architecture. More details in Section 3.1 [Default FDE Type Interpretation], page 15.
`SFRAME_FDE_TYPE_FLEX`	1	The flexible FDE type. Used for complex cases such as stack realignment (DRAP), non-standard CFA base registers, or when RA/FP recovery requires dereferencing or non-CFA base registers. The variable-length array of bytes may be interpreted as pairs of Control Data and Offset Data (or Padding Data), allowing for complex recovery rules (e.g., DRAP on AMD64, Stack Realignment). More details in Section 3.2 [Flexible FDE Type Interpretation], page 17.

Currently, five bits are reserved in the `sfda_func_info2` for indicating SFrame FDE types. In future, other ABIs/architectures may add even arch-specific FDE types. Each distinct FDE type may define a different layout, encoding, and interpretation of the variable-length data words trailing each SFrame FRE.

2.3.3.3 The SFrame FRE Types

A real world application can have functions of size big and small. SFrame format defines three types of SFrame FRE entries to efficiently encode the stack trace information for such

a variety of function sizes. These representations vary in the number of bits needed to encode the start address offset in the SFrame FRE.

The following constants are defined and used to identify the SFrame FRE types:

Name	Value	Description
`SFRAME_FRE_TYPE_ADDR1`	0	The start address offset (in bytes) of the SFrame FRE is an unsigned 8-bit value.
`SFRAME_FRE_TYPE_ADDR2`	1	The start address offset (in bytes) of the SFrame FRE is an unsigned 16-bit value.
`SFRAME_FRE_TYPE_ADDR4`	2	The start address offset (in bytes) of the SFrame FRE is an unsigned 32-bit value.

A single function must use the same type of SFrame FRE throughout. The identifier to reflect the chosen SFrame FRE type is stored in the `fre_type` bits in the SFrame FDE info byte, See Section 2.3.3 [The SFrame FDE Info Bytes], page 9.

2.4 SFrame FRE

The SFrame frame row entry sub-section contains the core of the stack trace information. An SFrame frame row entry (FRE) is a self-sufficient record containing SFrame stack trace information for a range of contiguous (instruction) addresses, starting at the specified offset from the start of the function.

Each SFrame FRE encodes the information to recover the CFA, FP and RA (as specified by the ABI or the FDE type) for the respective instruction addresses. To encode this information, each SFrame FRE is followed by S*N bytes, where:

- `S` is the size of each data word in the variable-length array of data words trailing the SFrame FRE, and
- `N` is the number of data words trailing the SFrame FRE.

NB: The term 'data word' is used throughout this specification in a colloquial sense to denote a discrete unit of information within an SFrame Frame Row Entry (FRE). It is intended to describe the semantic role of the data rather than its physical size. Consequently, 'data word' should not be understood to correlate with the architectural machine word size or any specific hardware data width; the actual size of a data word in the SFrame format is variable and is defined in the SFrame FRE info byte.

The entities `S`, `N` are encoded in the SFrame FRE info byte, via the `fre_dataword_size` and the `fre_dataword_count` respectively. More information about the precise encoding and range of values for `S` and `N` is provided later in the Section 2.4.1 [The SFrame FRE Info Word], page 13.

It is important to underline here that although the canonical interpretation of these data words is as stack offsets (to recover CFA, FP and RA) for default FDE type, these bytes *may* be used by future ABIs/architectures to convey other information on a per SFrame FRE basis.

In summary, SFrame file format, by design, supports a variable length array of bytes at the tail end of each SFrame FRE. To keep the SFrame file format specification flexible

yet extensible, the interpretation of these bytes is specific to ABI/arch or FDE type. More details about the precise interpretation are covered in the section Chapter 3 [Interpretation of SFrame FREs], page 15.

Next, the definitions of the three SFrame FRE types are as follows:

```
typedef struct sframe_frame_row_entry_addr1
{
  uint8_t sfre_start_address;
  sframe_fre_info sfre_info;
} ATTRIBUTE_PACKED sframe_frame_row_entry_addr1;

typedef struct sframe_frame_row_entry_addr2
{
  uint16_t sfre_start_address;
  sframe_fre_info sfre_info;
} ATTRIBUTE_PACKED sframe_frame_row_entry_addr2;

typedef struct sframe_frame_row_entry_addr4
{
  uint32_t sfre_start_address;
  sframe_fre_info sfre_info;
} ATTRIBUTE_PACKED sframe_frame_row_entry_addr4;
```

For ensuring compactness, SFrame frame row entries are stored unaligned on disk. Appropriate mechanisms need to be employed, as necessary, by the serializing and deserializing entities, if unaligned accesses need to be avoided.

`sfre_start_address` is an unsigned 8-bit/16-bit/32-bit integral field denoting the start address of a range of program counters, for which the SFrame FRE applies. The value encoded in the `sfre_start_address` field is the offset in bytes of the range's start address, from the start address of the function.

Further SFrame FRE types may be added in future.

2.4.1 The SFrame FRE Info Word

The SFrame FRE info byte is a bitfield split into four parts. From MSB to LSB:

Bit offset	Name	Description
7	`fre_mangled_ra_p`	Indicate whether the return address is mangled with any authorization bits (signed RA).
5-6	`fre_dataword_size`	Size of data word in bytes. Valid values are: `SFRAME_FRE_DATAWORD_1B`, `SFRAME_FRE_DATAWORD_2B`, and `SFRAME_FRE_DATAWORD_4B`.

1-4	`fre_dataword_count`	Being a 4-bit sized field, a max value of 15 is allowed. Typically, a value of up to 3 is sufficient for most ABIs to track all three of CFA, FP and RA. A value of zero indicates that the return address (RA) is undefined. A stack tracer may use this as indication that an outermost frame has been reached and the stack trace is complete.
0	`fre_cfa_base_reg_id`	Distinguish between SP or FP based CFA recovery.

Name	**Value**	**Description**
`SFRAME_FRE_DATAWORD_1B`	0	All data words following the fixed-length FRE structure are 1 byte long.
`SFRAME_FRE_DATAWORD_2B`	1	All data words following the fixed-length FRE structure are 2 bytes long.
`SFRAME_FRE_DATAWORD_4B`	2	All data words following the fixed-length FRE structure are 4 bytes long.

3 Interpretation of SFrame FREs

Each SFrame Frame Row Entry (FRE) provides information about a PC range within some function, encoded using a variable number of bytes (see Section 2.4 [SFrame Frame Row Entries], page 12). The interpretation of these bytes depends on the FDE type used to represent stack tracing information for the function.

3.1 Default FDE Type Interpretation

If the FDE type is `SFRAME_FDE_TYPE_DEFAULT`, the interpretation of the FRE bytes is ABI/arch-specific. Typically, these bytes are interpreted as a sequence of (signed integer) stack offsets.

The following sections describe the specific interpretation rules for currently supported architectures.

3.1.1 AMD64

Irrespective of the ABI, the first stack offset is always used to locate the CFA, by interpreting it as: CFA = `BASE_REG` + offset1. The identification of the `BASE_REG` is done by using the `fre_cfa_base_reg_id` field in the SFrame FRE info byte.

In AMD64, the return address (RA) is always saved on stack when a function call is executed. Further, AMD64 ABI mandates that the RA be saved at a `fixed offset` from the CFA when entering a new function. This means that the RA does not need to be tracked per SFrame FRE. The fixed offset is encoded in the SFrame file format in the field `sfh_cfa_fixed_ra_offset` in the SFrame header. See Section 2.2 [SFrame Header], page 5.

Hence, the second stack offset (in the SFrame FRE), when present, will be used to locate the FP, by interpreting it as: FP = CFA + offset2.

Hence, in summary:

Offset ID	**Interpretation in AMD64**
1	CFA = `BASE_REG` + offset1
2	FP = CFA + offset2

3.1.2 AArch64

Irrespective of the ABI, the first stack offset is always used to locate the CFA, by interpreting it as: CFA = `BASE_REG` + offset1. The identification of the `BASE_REG` is done by using the `fre_cfa_base_reg_id` field in the SFrame FRE info byte.

In AArch64, the AAPCS64 standard specifies that the Frame Record saves both FP and LR (a.k.a the RA). However, the standard does not mandate the precise location in the function where the frame record is created, if at all. Hence the need to track RA in the SFrame stack trace format. As RA is being tracked in this ABI, the second stack offset is always used to locate the RA, by interpreting it as: RA = CFA + offset2. The third stack offset will be used to locate the FP, by interpreting it as: FP = CFA + offset3.

Given the nature of things, the number of stack offsets seen on AArch64 per SFrame FRE is either 1 or 3.

Hence, in summary:

Offset ID	Interpretation in AArch64
1	CFA = `BASE_REG` + offset1
2	RA = CFA + offset2
3	FP = CFA + offset3

3.1.3 s390x

A stack tracer implementation must initialize the SP to the designated SP register value, the FP to the preferred FP register value, and the RA to the designated RA register value in the topmost stack frame of the callchain. This is required, as either the SP or FP is used as CFA base register and as the FP and/or RA are not necessarily saved on the stack. For RA this may only be the case in the topmost stack frame of the callchain. For FP this may be the case in any stack frame.

Irrespective of the ABI, the first stack offset is always used to locate the CFA. On s390x the value of the offset is stored adjusted by the s390x-specific `SFRAME_S390X_CFA_OFFSET_ADJUSTMENT` and scaled down by the s390x-specific `SFRAME_S390X_CFA_OFFSET_ALIGNMENT_FACTOR`, to enable and improve the use of signed 8-bit offsets on s390x. s390x-specific helpers `SFRAME_V2_S390X_CFA_OFFSET_ENCODE` and `SFRAME_V2_S390X_CFA_OFFSET_DECODE` are provided to perform or undo the adjustment and scaling. The CFA offset can therefore be interpreted as: CFA = `BASE_REG` + offset1 - `SFRAME_S390X_CFA_OFFSET_ADJUSTMENT` or CFA = `BASE_REG` + (offset1 * `SFRAME_S390X_CFA_OFFSET_ALIGNMENT_FACTOR`) - `SFRAME_S390X_CFA_OFFSET_ADJUSTMENT`. The identification of the `BASE_REG` is done by using the `fre_cfa_base_reg_id` field in the SFrame FRE info byte.

The (64-bit) s390x ELF ABI does not mandate the precise location in a function where the return address (RA) and frame pointer (FP) are saved, if at all. Hence the need to track RA in the SFrame stack trace format. As RA is being tracked in this ABI, the second stack offset is always used to locate the RA stack slot, by interpreting it as: RA = CFA + offset2, unless the offset has a value of `SFRAME_FRE_RA_OFFSET_INVALID`. RA remains unchanged, if the offset is not available or has a value of `SFRAME_FRE_RA_OFFSET_INVALID`. Stack tracers are recommended to validate that the `"unchanged RA"` pattern, when present, is seen only for the topmost stack frame. The third stack offset is used to locate the FP stack slot, by interpreting it as: FP = CFA + offset3. FP remains unchanged, if the offset is not available.

In leaf functions the RA and FP may be saved in other registers, such as floating-point registers (FPRs), instead of being saved on the stack. To represent this in the SFrame stack trace format, SFrame FDE of type `SFRAME_FDE_TYPE_FLEX` may be used.

Given the nature of things, for default type FDEs, the number of stack offsets seen on s390x per SFrame FRE is either 1, 2, or 3.

Hence, in summary:

Offset ID	Interpretation in s390x
1	CFA = `BASE_REG` + offset1
2	RA stack slot = CFA + offset2
	RA not saved if (offset2 == `SFRAME_FRE_RA_OFFSET_INVALID`)
3	FP stack slot = CFA + offset3

The s390x ELF ABI defines the CFA as stack pointer (SP) at call site +160. The SP can therefore be obtained using the SP value offset from CFA `SFRAME_S390X_SP_VAL_OFFSET` of -160 as follows: SP = CFA + `SFRAME_S390X_SP_VAL_OFFSET`

Future ABIs must specify the algorithm for identifying the appropriate SFrame FRE stack offsets in this chapter. This should inevitably include the blueprint for interpreting the variable number of bytes at the tail end of the SFrame FRE for the specific ABI/arch.

3.2 Flexible FDE Type Interpretation

Flexible FDEs (`SFRAME_FDE_TYPE_FLEX`) are used in cases where the most common default recovery rules implied by `SFRAME_FDE_TYPE_DEFAULT` are insufficient. Common use cases include:

- DRAP (Dynamically Realigned Argument Pointer): Where the CFA is based on a register other than SP or FP, or requires dereferencing.
- Stack Realignment: Where strict alignment requirements (e.g., AVX512) force dynamic stack adjustments.
- Register-based RA/FP Locations: Where the Return Address or Frame Pointer is transiently saved in a general-purpose register and/or requires a dereference rule.

For flexible FDE types, the variable-length bytes trailing an SFrame FRE can be interpreted as one of the following:

1. Control Data: Encodes the base register number, a dereference flag, and a register-mode flag. A value of 0 is reserved as the *padding data word*.
2. Offset Data: Encodes the signed offset to be added to the base.

For each tracked entity (CFA, RA, FP), the SFrame FRE carries a pair of data words to specify the respective recovery rule. The pair of data words appear in the order: CFA, RA, FP. These data words obey the `fre_dataword_size` defined in the FRE info byte (i.e., they are 1, 2, or 4 bytes wide).

Given the nature of things, since CFA is always tracked, the first two data words pertain to CFA recovery. If RA recovery rule is unspecified (because the RA can be recovered from its default location), a single padding data word is used instead of the pair of Control data word and Offset data word if FP recovery rule is to be specified using the subsequent data words.

Following is the order of information for specifying the recovery rule for a tracked entity in a flexible FDE.

Encoding of Data Word 1 (Control Data)

The first data word of the pair is an unsigned integer of size `fre_dataword_size`. It is used as a bitfield that describes register/control data for the tracked entity. From LSB to MSB:

Bit Offset	Name	Description
0	`reg_p`	Register-based Location Rule If 1, the base is a DWARF register (encoded in bits 3+). If 0, the base is the CFA (used for RA/FP recovery).

1	`deref_p`	Dereference Flag If 1, the location of the value is the address (`Base + Offset`), i.e., value = `*(Base + Offset)`. If 0, the value is `Base + Offset`.
2	`unused`	Unused bit.
3+	`regnum`	The DWARF register number used as the base. Effective only if `reg_p` is 1.

A value of 0 (i.e., regnum = 0, deref_p = 0, reg_p = 0) in the Control Data Word is used to indicate that no further data words follow for the tracked entity. This is to convey an absence of recovery rule for the respective tracked entity (which means that fixed offsets `sfh_cfa_fixed_fp_offset` or `sfh_cfa_fixed_ra_offset` apply if used for the ABI/arch). Note that, using a value of 0 as padding data word, does mean that currently, e.g., for RA, the rule RA = CFA + 0 cannot be encoded. NB: RA = CFA + 0 is distinct from RA = *(CFA + 0). The former should not be needed for any ABI, and the latter is representable (regnum = 0, deref_p = 1, reg_p = 0).

Encoding of Data Word 2 (Offset Data)

The second data word of the pair is a signed integer of width `fre_dataword_size`. It is used as a offset for the respective tracked entity (CFA, FP or RA).

Recovery Rules

The value of the tracked entity (CFA, RA, or FP) is calculated using the following logic:

```
Base   = (reg_p == 1) ? Register[regnum] : CFA;
Addr   = Base + Offset2;
Value  = (deref_p == 1) ? *Addr : Addr;
```

Examples:

- CFA = *(RBP - 8): (Typical DRAP pattern on AMD64)
  Data Word 1: `(RBP << 3) | (1 << 1) | 1` (Reg RBP, deref_p=True, reg_p=True)
  Data Word 2: `-8`
- FP = *(RBP + 0):
  Data Word 1: `(RBP << 3) | (1 << 1) | 1` (Reg RBP, deref_p=True, reg_p=True)
  Data Word 2: `0`
- RA = *(CFA - 8): (Standard RA recovery on AMD64)
  Data Word 1: `(0 << 3 | (1 << 1) | 0)` (reg_p=False, implies Base=CFA, deref_p=True by implication of standard stack save)
  Data Word 2: `-8`

If the FDE type is `SFRAME_FDE_TYPE_FLEX`, the FRE bytes are interpreted using a universal encoding scheme designed to handle complex recovery rules (such as DRAP or non-standard RA locations).

Appendix A Generating Stack Traces using SFrame

Using some C-like pseudocode, this section highlights how SFrame provides a simple, fast and low-overhead mechanism to generate stack traces. Needless to say that for generating accurate and useful stack traces, several other aspects will need attention: finding and decoding bits of SFrame section(s) in the program binary, symbolization of addresses, to name a few.

In the current context, a `frame` is the abstract construct that encapsulates the following information:

- program counter (PC),
- stack pointer (SP), and
- frame pointer (FP)

With that said, establishing the first `frame` should be trivial:

```
// frame 0
frame->pc = current_IP;
frame->sp = get_reg_value (REG_SP);
frame->fp = get_reg_value (REG_FP);
```

where `REG_SP` and `REG_FP` are are ABI-designated stack pointer and frame pointer registers respectively.

Next, given frame N, generating stack trace needs us to get frame N+1. This can be done as follows:

```
// Get the PC, SP, and FP for frame N.
pc = frame->pc;
sp = frame->sp;
fp = frame->fp;
// Populate frame N+1.
int err = get_next_frame (&next_frame, pc, sp, fp);
```

where given the values of the program counter, stack pointer and frame pointer from frame N, `get_next_frame` populates the provided `next_frame` object and returns the error code, if any.

In the following pseudocode for `get_next_frame`, the `sframe_*` functions fetch information from the SFrame section. Note that the stack tracer must retrieve the FDE type to decide how to interpret the FRE data words.

```
fre = sframe_find_fre (pc, &fde_type);
if (fre && fde_type == SFRAME_FDE_TYPE_DEFAULT)
    // Whether the base register for CFA tracking is REG_FP.
    base_reg_val = sframe_fre_base_reg_fp_p (fre) ? fp : sp;
    // Get the CFA stack offset from the FRE.
    cfa_offset = sframe_fre_get_cfa_offset (fre);
    // Get the fixed RA offset or FRE stack offset as applicable.
    ra_offset = sframe_fre_get_ra_offset (fre);
    // Get the fixed FP offset or FRE stack offset as applicable.
    fp_offset = sframe_fre_get_fp_offset (fre);
```

```
cfa = base_reg_val + cfa_offset;
next_frame->sp = cfa [+ SFRAME_S390X_SP_VAL_OFFSET on s390x];

ra_stack_loc = cfa + ra_offset;
// Get the address stored in the stack location.
next_frame->pc = read_value (ra_stack_loc);

if (fp_offset is VALID)
    fp_stack_loc = cfa + fp_offset;
    // Get the value stored in the stack location.
    next_frame->fp = read_value (fp_stack_loc);
else
    // Continue to use the value of fp as it has not
    // been clobbered by the current frame yet.
    next_frame->fp = fp;
```

For SFrame FDE of type `SFRAME_FDE_TYPE_FLEX`, read the set of data words and apply the recovery rules accordingly.

```
if (fre && fde_type == SFRAME_FDE_TYPE_FLEX)
    // Get the base register, offset, and deref_p for CFA tracking.
    // The first FRE offset (index 0) is the CFA Control Data.
    cfa_reg_data = sframe_fre_get_offset (fre, 0);
    cfa_offset = sframe_fre_get_offset (fre, 1);

    // Get the RA reg, offset, and deref_p.
    // The third FRE data word (index 2) is the RA Control Data.
    ra_reg_data = sframe_fre_get_udata (fre, 2);
    if (ra_reg_data != SFRAME_FRE_RA_OFFSET_INVALID)
            ra_offset = sframe_fre_get_offset (fre, 3);
            fp_tracking_p = fre.num_offsets > 3;
            fp_data_index = 3;
    else
            fp_tracking_p = fre.num_offsets > 4;
            fp_data_index = 4;

    // Get the FP reg, offset, and deref_p (if present).
    if (fp_tracking_p)
         fp_reg_data = sframe_fre_get_udata (fre, fp_data_index);
         fp_offset = sframe_fre_get_fp_offset (fre);

    // Safety check for topmost frames:
    // If recovery requires non-standard registers (not SP/FP),
    // it is only valid if we are at the top of the stack
    // (where those registers haven't been clobbered).
    cfa_base_reg = SFRAME_V3_FLEX_FDE_OFFSET_REG_NUM (cfa_reg_data);
    if (!topmost_frame_p && (cfa_base_reg != REG_FP
                             && cfa_base_reg != REG_SP))
```

```
            return ERR_SFRAME_UNSAFE_UNWIND;

        // Apply rules to recover CFA and RA
        cfa = sframe_apply_rule (cfa_reg_data, cfa_offset, cfa, 1);
        ra = sframe_apply_rule (ra_reg_data, ra_offset, cfa, 0);

        if (fp_tracking_p)
            next_frame->fp
               = sframe_apply_rule (fp_reg_data, fp_offset, cfa, 0);
        else
            next_frame->fp = fp;

        next_frame->sp = cfa;
        next_frame->pc = ra;
    else
        ret = ERR_NO_SFRAME_FRE;
```

The `sframe_apply_rule` helper function abstracts the logic of interpreting the Control Data and Offset Data pair for flexible FDEs:

```
// Apply SFrame V3 Flex FDE recovery rule.
// reg_data: The Control Data (Data word 1)
             containing reg_p, deref_p, regnum.
// offset:   The Offset (Data word 2).
// cfa:      The current CFA value (used as base if reg_p is 0).
// cfa_p:    Bool indicating if we are currently recovering the
             CFA itself.

sframe_apply_rule (reg_data, offset, cfa, cfa_p)
    reg_p = SFRAME_V3_FLEX_FDE_OFFSET_REG_P (reg_data);

    // Determine Base Address:
    // If reg_p is set, read from the specific DWARF register.
    // If reg_p is clear, use the CFA (unless we are recovering the
    // CFA itself, in which case reg_p MUST be set).
    if (reg_p)
         reg_num = SFRAME_V3_FLEX_FDE_OFFSET_REG_NUM (reg_data);
         base_loc = get_reg_value (reg_num);
    else
         base_loc = cfa;

    // CFA recovery must always specify a base register.
    assert (!cfa_p || reg_p);

    // Add the displacement
    loc = base_loc + offset;

    // Dereference if required
```

```
deref_p = SFRAME_V3_FLEX_FDE_OFFSET_REG_DEREF_P (reg_data);
value = deref_p ? read_value (loc) : loc;

return value;
```

Index