BEDROCK ARCHITECTURE

Bedrock ELF ABI

ELF Object, Linking, and Loading ABI

Draft

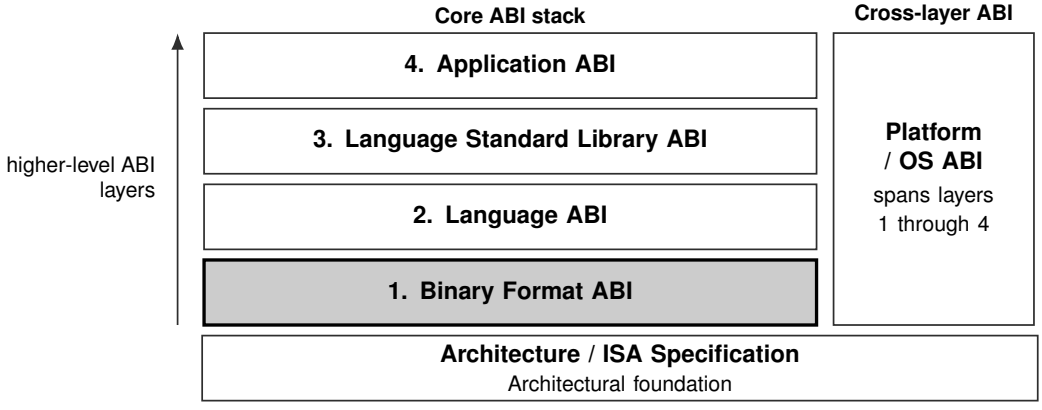
CONTENTS

LIST OF TABLES

1 SCOPE

This document defines the Bedrock binary-format contract for ELF objects, linking, relocation, loading, dynamic linking, TLS metadata, and code models. The Bedrock ISA Specification supplies the referenced architectural objects. Language bindings and application interfaces each own their higher-level contracts.

1.1 Contract Position



Current document: Bedrock ELF ABI. The shaded block identifies its primary position in the architecture and ABI stack. The Platform / OS ABI spans and constrains the four core ABI layers; the ISA specification defines the architecture beneath them.

1.2 Profile

Profile: `bedrock-elf`

2 TERMINOLOGY

These terms are used by the Bedrock ELF ABI. ISA-level address, register, and effective-address terms keep the meanings defined by the architecture manual.

ELF ABI The ELF, relocation, dynamic-linking, TLS, and code-model contract shared by language bindings and loaders.

pre-segment virtual address
The stable coordinate used by ELF symbols, relocations, ordinary function addresses, and ordinary data pointers before segment pre-translation.

load base The runtime address bias applied to a loaded ET_DYN object before its symbol values and relative relocations are interpreted.

ABI-visible linkage domain
The region in which ordinary ELF code addresses are near-call entry points represented solely by pre-segment coordinates.

near-call entry address
A pre-segment code address that can be reached with ordinary CALL and returned from with ordinary RET inside the current linkage domain.

veneer A near-call entry point supplied by a linker or loader to encapsulate a more complex transfer sequence for ordinary ABI callers.

code model A set of placement and relocation assumptions used by code generation to choose direct displacement forms, GOT/PLT forms, or full-width address materialization.

3 ELF OBJECT FORMAT

Bedrock objects use the 64-bit little-endian ELF format and machine identifier `EM_BEDROCK` (`0xffb0`). A producer may emit relocatable objects, executable images, or shared/position-independent images using `ET_REL`, `ET_EXEC`, or `ET_DYN`.

All relocation sections use `Elf64_Rela` entries with explicit addends and the `.rela` section-name stem. Ordinary symbol and string tables use `.symtab` and `.strtab`. The baseline value of `e_flags` is zero; a consumer accepts zero as the baseline profile and rejects every unknown nonzero value.

3.1 ELF Identification Bytes

Table 3-1. ELF e_ident Values

e_ident Field	Value
`EI_MAG`	`0x7f 'E' 'L' 'F'`
`EI_CLASS`	`ELFCLASS64`
`EI_DATA`	`ELFDATA2LSB`
`EI_VERSION`	`EV_CURRENT`
`EI_OSABI`	`ELFOSABI_NONE`
`EI_ABIVERSION`	`0`

3.2 ELF Header Sizes

Table 3-2. ELF Header Size Values

Header Field	Bytes
`e_ehsize`	64
`e_phentsize`	56
`e_shentsize`	64

3.3 Entry Point Semantics

Table 3-3. ELF Entry Point Rules

File Type	e_entry Meaning
`ET_REL`	0
`ET_EXEC`	absolute pre-segment virtual byte address
`ET_DYN (executable)`	load-bias-relative pre-segment virtual byte address; runtime entry is load base + e entry
`ET_DYN (shared object, entry value 0)`	0

3.4 Program Header Policy

The standard `PT_LOAD`, `PT_DYNAMIC`, `PT_INTERP`, `PT_NOTE`, and `PT_TLS` program headers retain their ordinary ELF meanings.

Program permission bits use the standard values `PF_X=1`, `PF_W=2`, and `PF_R=4`. Every `PT_LOAD` has `p_align` equal to the selected page size or a larger power-of-two alignment; the minimum supported page alignment is 4096 bytes.

3.5 Section and Symbol Policy

Standard ELF section types and the `SHF_WRITE`, `SHF_ALLOC`, `SHF_EXECINSTR`, and `SHF_TLS` flags retain their generic meanings. Symbol binding and type occupy `st_info` in the ordinary ELF encoding. Architecture-specific `st_other` bits are reserved and must be zero.

In an `ET_REL` object, `st_value` is a byte offset from the start of the defining section. After final link, it is the symbol's pre-segment virtual byte address. Ordinary function symbols retain their pre-segment address representation, and ordinary object symbols retain their byte-address representation.

4 PROGRAM LOADING

Bedrock ELF program loading follows the ordinary ELF segment model. This ELF ABI defines executable mapping, loader-visible object rules, and entry-image state.

4.1 Loading Rules

The loader may execute `ET_EXEC` and executable `ET_DYN` images. It honors `PT_INTERP`, maps each loadable segment at its declared page alignment, and rejects an alignment below 4096 bytes. For a segment whose `p_memsz` exceeds `p_filesz`, the loader fills every byte in the half-open range $[\texttt{p\_vaddr} + \texttt{p\_filesz}, \texttt{p\_vaddr} + \texttt{p\_memsz})$ with zero. Object and section alignment requirements remain binding. The zero-filled gap may begin at any byte alignment.

4.2 Page Permissions

Every `PT_LOAD` shall set `PF_R`. A loader rejects a `PT_LOAD` whose flags are zero or pair `PF_W` or `PF_X` with a clear `PF_R`. For an accepted segment, its flags select one Normal leaf AM as shown below. The loader rejects `PF_R|PF_W|PF_X` because the available leaf AMs separate writable and executable mappings.

Table 4-1. ELF Segment Permission Mapping

ELF Flags	Leaf AM and Fields	Meaning
`PF_R`	`AM=R`, P, U	readable user page
`PF_R|PF_W`	`AM=RW`, P, U	readable writable user page
`PF_R|PF_X`	`AM=RX`, P, U	readable executable user page

4.3 Execution Environment

The ELF contract interprets `e_entry`, `p_vaddr`, symbol values, relocation places, and linked code and data addresses in one stable pre-segment virtual-address coordinate. Every ABI-permitted path that consumes one such coordinate preserves its arithmetic and linkage meaning and resolves an actual access to the intended byte of the intended object. At transfer to `e_entry`, the code context must permit fetch from the entry address, the data context must cover the mapped ordinary data regions, and the stack context must permit reads and writes to the entry stack.

The loader maps each post-segment linear range required by these coordinates to the intended physical pages with the AM and U permissions required by the ELF image. Process entry, dynamic linking, module transitions, TLS paths, unwind and debugger state, JIT publication, and shared pointer conventions preserve the same coordinate invariants.

Executable images and ordinary shared objects form one ABI-visible near-call linkage domain. Their function symbols use pre-segment code addresses. The ordinary dynamic-linking contract covers near calls, and a linker-supplied veneer brings a segment-changing transfer into that domain.

4.4 Program Entry State

Before transferring control, the loader completes image relocations and publishes executable bytes. At transfer, `PC` equals the run-time value of `e_entry`; `SP` names a writable entry stack with 16-byte alignment; and the code, data, and stack segment contexts provide the image described above. A TLS image uses `GS0` as its thread-pointer base.

The loader clears `FLAGS`, `FSTATUS`, and `FFLAGS`. General, floating-point, vector, and predicate registers outside the stated entry contract carry unspecified values. An external process-entry ABI defines the entry-stack payload and the interpretation of initial arguments. This language-independent state permits an ELF entry point to establish the run-time convention selected by its implementation language.

Table 4-2. Language-Independent ELF Program Entry State

Property	Contract
Entry PC	`PC = e_entry`
Entry stack	`SP, 16-byte aligned, read/write`
Segment contexts	`CS/DS/SS`
TLS base	`GS0`
Readiness	relocations complete, executable bytes published
Cleared state	`FLAGS, FFLAGS, FSTATUS`
Stack payload owner	`external-process-entry-abi`

5 SECTIONS AND SYMBOLS

Bedrock uses ordinary ELF sections with the usual ELF meanings.

5.1 Symbol Model

Bedrock supports the standard local, global, and weak bindings and the default, hidden, and protected visibility classes. The standard notype, object, function, TLS, ifunc, section, and file symbol types retain their ELF meanings.

- Function symbols name the first instruction of the callable entry point.
- Object symbols name the first byte of the object storage.
- Symbol values follow the Section and Symbol Policy in Section 3.5.

6 RELOCATION SET

Bedrock ELF objects use RELA relocation entries with explicit addends. The Bedrock ABI defines how each relocation type patches instruction or data fields.

6.1 Expression Model

Every static relocation starts with the canonical expression `symbol + addend`, where the symbol and explicit RELA addend are interpreted in the address space of the relocation family. Ordinary relocations use ABI symbol addresses, and TLS relocations use GS0-relative offsets.

Absolute and section-relative results are permitted when the selected relocation type defines them. PC-relative address materialization subtracts `place`, the address of the patched field. Direct branch and call payloads subtract `next_pc`, the address of the instruction following the patched instruction. An out-of-range result for the signedness and width stated in the relocation table is a link error, and the linker rejects the relocation.

For a PC-relative address-materialization relocation, let `I` be the address of byte zero of the containing instruction and let $\delta = \texttt{place} - \texttt{I}$ be the relocation field's byte offset. Let A_{src} be the addend in the source expression before field-offset bias. The encoded value shall equal the target plus A_{src} minus `I`. Because the relocation-table expressions subtract `place`, the assembler records the ELF addend as $\texttt{addend} = A_{\mathrm{src}} + \delta$ for `PCREL`, `GOTPCREL`, `GOT_BASE_PCREL`, and `TLSDESC_GOTPCREL` relocations. The assembler supplies every field offset through the addend. The PLT addend described in Section 7.3 is one instance of this general rule.

6.2 Relocation Metasyntax Vocabulary

Table 6-1. Relocation Metasyntax Vocabulary

Identifier or function	Meaning
`symbol`	Runtime byte address of the symbol named by the relocation after symbol resolution.
`addend`	Explicit addend stored in the ELF64 Rela entry.
`place`	Runtime byte address of the relocation field being patched.
`next_pc`	Runtime byte address of the next instruction after the instruction being patched.
`section_base`	Runtime byte address of the beginning of the symbol's defining section.
`symbol_size`	Byte size of the resolved symbol.
`load_base`	Runtime load base address of the loaded object that owns the relocation.
`got_base`	Runtime byte address of the global offset table base for the loaded object.
`got(symbol)`	Runtime byte address of the GOT slot allocated for `symbol`.
`plt(symbol)`	Runtime byte address of the PLT entry allocated for `symbol`.
`tls(symbol)`	Byte offset of the TLS symbol from the ABI TLS base in GS0.
`tlsdesc(symbol)`	Runtime byte address of the TLSDESC entry allocated for the TLS symbol.
`resolver(x)`	Runtime byte address returned by calling the resolver function at address `x`.
`copy(symbol, symbol_size)`	Copy operation over the resolved symbol's bytes.
`tls_descriptor(symbol)`	Adjacent function and argument words selected for the TLS symbol.

6.3 GOT and PLT Model

The ordinary GOT stores 64-bit addresses for position-independent code and dynamic binding. A GOT relocation allocates or references the symbol's slot in the loaded object. The PLT contains executable near-call entries; a PLT relocation names that entry as its callable target. Every ordinary PLT slot is resolved before application entry or constructors execute.

GOT and PLT function addresses stay inside the ABI-visible near-call linkage domain.

6.4 Relocations

Table 6-2. Bedrock ELF Relocation Types

Type	Name	Size	Calculation
0	R_BEDROCK_NONE	0	0
1	R_BEDROCK_ABS8	8-bit unsigned	symbol + addend
2	R_BEDROCK_ABS16	16-bit unsigned	symbol + addend
3	R_BEDROCK_ABS32S	32-bit signed	symbol + addend
4	R_BEDROCK_ABS64	64-bit unsigned	symbol + addend
5	R_BEDROCK_IMM8S	8-bit signed	symbol + addend
6	R_BEDROCK_IMM16S	16-bit signed	symbol + addend
7	R_BEDROCK_IMM32S	32-bit signed	symbol + addend
8	R_BEDROCK_IMM64	64-bit unsigned	symbol + addend
9	R_BEDROCK_DISP8S	8-bit signed	symbol + addend
10	R_BEDROCK_DISP16S	16-bit signed	symbol + addend
11	R_BEDROCK_DISP32S	32-bit signed	symbol + addend
12	R_BEDROCK_DISP64	64-bit unsigned	symbol + addend
13	R_BEDROCK_PCREL8S	8-bit signed	symbol + addend - place
14	R_BEDROCK_PCREL16S	16-bit signed	symbol + addend - place
15	R_BEDROCK_PCREL32S	32-bit signed	symbol + addend - place
16	R_BEDROCK_PCREL64	64-bit signed	symbol + addend - place
17	R_BEDROCK_BRDISP8S	8-bit signed	symbol + addend - next_pc
18	R_BEDROCK_BRDISP16S	16-bit signed	symbol + addend - next_pc
19	R_BEDROCK_BRDISP32S	32-bit signed	symbol + addend - next_pc
20	R_BEDROCK_CALL16S	16-bit signed	symbol + addend - next_pc
21	R_BEDROCK_CALL32S	32-bit signed	symbol + addend - next_pc
22	R_BEDROCK_SECTION_REL32	32-bit unsigned	symbol + addend - section_base
23	R_BEDROCK_SECTION_REL64	64-bit unsigned	symbol + addend - section_base
24	R_BEDROCK_GOT64	64-bit unsigned	got(symbol) + addend
25	R_BEDROCK_GOTPCREL32S	32-bit signed	got(symbol) + addend - place
26	R_BEDROCK_GOTPCREL64	64-bit signed	got(symbol) + addend - place
27	R_BEDROCK_GOTOFF32S	32-bit signed	symbol + addend - got_base
28	R_BEDROCK_GOTOFF64	64-bit signed	symbol + addend - got_base
29	R_BEDROCK_GOT_BASE_PCREL32S	32-bit signed	got_base + addend - place
30	R_BEDROCK_GOT_BASE_PCREL64	64-bit signed	got_base + addend - place

Type	Name	Size	Calculation
31	R_BEDROCK_PLT16S	16-bit signed	plt(symbol) + addend - next_pc
32	R_BEDROCK_PLT32S	32-bit signed	plt(symbol) + addend - next_pc
33	R_BEDROCK_PLT64	64-bit unsigned	plt(symbol) + addend
34	R_BEDROCK_RELATIVE	64-bit unsigned	load_base + addend
35	R_BEDROCK_GLOB_DAT	64-bit unsigned	symbol + addend
36	R_BEDROCK_JUMP_SLOT	64-bit unsigned	symbol + addend
37	R_BEDROCK_COPY	variable-size	copy(symbol, symbol_size)
38	R_BEDROCK_IRELATIVE	64-bit unsigned	resolver(load_base + addend)
39	R_BEDROCK_TLS_OFFSET32S	32-bit signed	tls(symbol) + addend
40	R_BEDROCK_TLS_OFFSET64	64-bit signed	tls(symbol) + addend
41	R_BEDROCK_TLSDESC_GOTPCREL32S	32-bit signed	tlsdesc(symbol) + addend - place
42	R_BEDROCK_TLSDESC_GOTPCREL64	64-bit signed	tlsdesc(symbol) + addend - place
43	R_BEDROCK_TLSDESC_CALL	0	0
44	R_BEDROCK_TLSDESC	128-bit pair	tls_descriptor(symbol)

6.5 Relocation Families and Additional Constraints

Relocations 1–12 write absolute data, immediate, or effective-address payloads. Relocations 13–23 write PC-relative, next-instruction-relative, or section-relative results. Relocations 24–33 address GOT slots, the GOT base, symbols relative to that base, and PLT entries. Relocations 34–38 are ordinary dynamic-loader operations. Relocations 39–44 implement GS0 local-exec and TLSDESC TLS. The table defines widths and calculations; the dynamic-linking and TLS sections supply additional rules.

7 DYNAMIC LINKING

Dynamic objects use a fixed eager-binding Bedrock GOT and PLT contract.

7.1 Dynamic Linking Rules

Ordinary dynamic linking supports standard and indirect functions, copy relocations, and R_BEDROCK_IRELATIVE. The loader resolves every ordinary R_BEDROCK_JUMP_SLOT and R_BEDROCK_IRELATIVE relocation before transferring control to the program entry point or any constructor. Shared-object calls use near CALL/RET within the ABI-visible linkage domain. The GOT base symbol is _GLOBAL_OFFSET_TABLE_.

Every ordinary PLT entry occupies 32 bytes and has 16-byte alignment.

7.2 GOT Reserved Entries

Table 7-1. Global Offset Table Reserved Entries

Index	Name	Meaning
0	_DYNAMIC	dynamic section address
1	reserved	zero
2	reserved	zero

7.3 PLT Layout

Table 7-2. Procedure Linkage Table Layout

Offset	Bytes or field	Effect
+0x00	e7 c9 80 67	JMP.Q [PC + disp64]
+0x04	disp64	little-endian R_BEDROCK_GOTPCREL64 relocation field
+0x0c	twenty 01 bytes	canonical NOP padding through offset +0x1f

A PLT relocation uses explicit addend +4. Architectural PC names byte zero of the current instruction, whereas relocation place P is the field at entry offset four; therefore GOT(S)+A-P evaluates to GOT(S)-PLTentry, the displacement consumed by the PC-relative EA. The referenced .got.plt slot is 8-byte aligned and contains the eagerly resolved 64-bit near target. Both the PLT entry and its slot must be reachable under the active CS bounds used by that PC-relative EA.

A PLT transfer preserves R0 through R7, F0 through F7, GS1 through GS5, FLAGS, and all segment state. The loader writes each final .got.plt value before publication. Published slots are immutable and may be covered by PT_GNU_RELRO.

8 THREAD-LOCAL STORAGE

Thread-local storage is addressed through the GS0 segment register. TLS references that remain inside the program and have a link-time constant offset use the local-exec model. TLS references that cross dynamic object boundaries, may be interposed, or otherwise require runtime resolution use the TLSDESC model.

8.1 TLS Base Model

GS0 is the TLS base register. A TLS offset is translated by GS0 segment pre-translation before page-table translation. The local-exec model encodes a link-time constant byte offset from the GS0 base; TLSDESC supplies runtime resolution when the reference crosses a dynamic-object boundary or remains interposable.

Table 8-1. TLS Relocation Families

Model	Relocations
local-exec	`R_BEDROCK_TLS_OFFSET32S`
	`R_BEDROCK_TLS_OFFSET64`
TLSDESC	`R_BEDROCK_TLSDESC_GOTPCREL32S`
	`R_BEDROCK_TLSDESC_GOTPCREL64`
	`R_BEDROCK_TLSDESC_CALL`
	`R_BEDROCK_TLSDESC`

Taking the address of a TLS object returns an ordinary pre-segment pointer to that thread instance. Every ABI-permitted GS0 and ordinary-data path consuming that coordinate resolves it to the same byte of that TLS object. Copying a TLS pointer to another thread retains its originating thread instance; the pointer becomes invalid when its originating thread terminates or the defining shared object is unloaded.

8.2 Model Selection

- A TLS symbol defined in the same program image may use local-exec addressing when its byte offset from GS0 is link-time constant.
- A TLS reference from a shared object or to an interposable TLS symbol uses TLSDESC.
- The linker may relax a TLSDESC sequence to local-exec when symbol binding and final placement make the GS0-relative offset constant.
- A source expression `S+k` uses the descriptor for `S`; after resolution, the caller adds `k` to the returned offset. The descriptor relocations encode the base symbol `S`.

8.3 TLSDESC Entry

Table 8-2. TLSDESC Entry Layout

Offset	Field	Type	Meaning
+0x00	function	u64	near-call address of the resolver or fast function
+0x08	argument	u64	signed direct GS0-relative offset or resolver-specific opaque token

Each entry occupies 16 bytes in .got and has 16-byte alignment. The loader resolves the symbol binding, writes both words, and completes any memory ordering required to publish the pair before application code can observe it. Both words are immutable after publication and may subsequently be protected by PT_GNU_RELRO. The resolver and fast function may use loader-private near-call addresses.

The argument word belongs to the implementation of the selected function. A fast function may load [R0+8] and return it directly. An opaque token is loader-owned metadata and remains valid while the object that defines the TLS symbol is loaded. A resolver may perform dynamic per-thread lookup or allocation. The published descriptor remains immutable for both resolver and fast-function paths.

8.4 TLSDESC Call ABI

The canonical sequence is exactly:

```
LEA.Q [PC + descriptor@TLSDESC_GOTPCREL], R0
CALL  [R0]
```

The LEA.Q places the descriptor address in R0. CALL [R0] reads the 64-bit near-call target from descriptor offset zero and preserves R0, so the called function receives the descriptor address in R0. The canonical sequence places the LEA.Q and CALL adjacently.

The function returns a valid signed byte offset from the GS0 TLS base in R0. It preserves GS0, R8–R15, F8–F15, V16–V31, and P8–P15. The linkage protocol catalog identifies the resolver scratch set. Zero is a valid TLS offset. Resolution failure terminates the resolver path.

8.5 TLSDESC Relocations

Descriptor address. R_BEDROCK_TLSDESC_GOTPCREL32S and R_BEDROCK_TLSDESC_GOTPCREL64 apply to the PC-relative displacement field of the canonical LEA.Q and compute the address of the descriptor for STT_TLS symbol S. The source expression has semantic TLS addend zero. Under the PC-relative field-bias rule in Section 6.1, the RELA addend is exactly $A = \delta$ and contains solely the instruction-field offset.

Call marker. R_BEDROCK_TLSDESC_CALL has size zero and names the same TLS symbol as the paired TLSDESC GOTPCREL relocation on the immediately preceding LEA.Q. Its addend is zero and its place is byte zero of the exact immediately following CALL [R0]. It is a static-link relaxation marker consumed exclusively by the static linker. A producer places labels and control-flow targets outside both instructions of a relaxable sequence.

Descriptor initialization. R_BEDROCK_TLSDESC applies only to STT_TLS symbols, has addend zero, and is placed at offset zero of an aligned descriptor. It directs the loader to initialize two adjacent 64-bit words with the selected {function, argument} pair for S; this operation initializes the pair directly. A link or load rejects an unresolved weak TLS symbol because GS0-relative offset zero represents a valid resolved offset. A weak symbol that resolves to a definition is handled normally.

The linker may merge descriptors only when they name the same resolved binding S and have the required zero semantic addend. If any unrelaxed use remains, a final dynamic link emits one aligned pair in .got and the corresponding dynamic R_BEDROCK_TLSDESC. The loader initializes the pair eagerly before publication. A fully static executable relaxes every TLSDESC use; any remaining use makes the static link fail. If all uses are relaxed, the final link discards the descriptor and its initialization relocation.

8.6 TLSDESC Local-Exec Relaxation

Relaxation is permitted only after final binding proves that S is non-preemptible and TLS(S) is a link-time constant. It replaces the entire canonical pair while preserving the section size and every following instruction address:

- With R_BEDROCK_TLSDESC_GOTPCREL32S, the original 7-byte LEA.Q and 4-byte CALL [R0] occupy 11 bytes. If TLS(S) fits signed 32 bits, the linker emits LEN 11, MOV.Q TLS(S), R0, applies R_BEDROCK_TLS_OFFSET32S with addend zero to its 32-bit immediate payload, and writes four zero padding bytes after the natural 7-byte encoding.
- With R_BEDROCK_TLSDESC_GOTPCREL64, the original 11-byte LEA.Q and 4-byte CALL [R0] occupy 15 bytes. The linker emits LEN 15, MOV.Q TLS(S), R0, applies R_BEDROCK_TLS_OFFSET64 with addend zero to its 64-bit immediate payload, and writes four zero padding bytes after the natural 11-byte encoding.

The linker consumes both code relocations. The removed call may eliminate normal caller-saved clobbers and unobservable resolver stack activity. Code relies only on the returned offset and defined preserved state. A descriptor is allocated exactly when an unrelaxed reference remains.

9 CODE MODELS

Code models define placement and locality assumptions that assemblers, linkers, and compilers may use when selecting address materialization, displacement, GOT, PLT, and relocation forms. Bedrock 32-bit displacement and direct control-transfer forms are signed, so direct 32-bit relative references cover a -2 GiB to +2 GiB - 1 window.

Table 9-1. Bedrock Code Models

Model	Placement contract	Default access strategy
`low`	static addresses in `0x00000000..0x7fffffff`	signed 32-bit absolute data and direct transfers
`small`	referenced code, data, GOT, and PLT fit one signed 32-bit relative window	PC-relative access; GOT/PLT for interposable symbols
`medium`	code, GOT, and PLT fit one signed 32-bit window; data placement is unrestricted	direct code; GOT-resolved 64-bit data addresses
`high`	static high image with internal references in a signed 32-bit displacement window	image-base or PC-relative access with full-width fallback
`large`	unrestricted 64-bit address space	full-width materialization and indirect transfer

9.1 low Model Details

The `low` model is a static image model whose named code and data lie between `0x00000000` and `0x7fffffff`. Code uses direct signed 32-bit branch or call payloads; data uses absolute or base-relative signed 32-bit payloads. The model permits omission of both GOT and PLT.

Its default relocation set is `R_BEDROCK_ABS32S`, `R_BEDROCK_IMM32S`, `R_BEDROCK_DISP32S`, `R_BEDROCK_BRDISP32S`, and `R_BEDROCK_CALL32S`. The linker may shrink a direct transfer or address materialization after final placement proves that the result fits a shorter encoding.

9.2 small Model Details

The `small` model is position-independent and assumes that local code, nearby data, the GOT, and the PLT fit the signed 32-bit relative window of the use site or selected base. Code uses direct signed 32-bit transfers; data uses PC-relative or GOTPCREL signed 32-bit materialization.

Its default relocations are `R_BEDROCK_BRDISP32S`, `R_BEDROCK_CALL32S`, `R_BEDROCK_GOTPCREL32S`, `R_BEDROCK_PLT32S`, and `R_BEDROCK_TLSDESC_GOTPCREL32S`. The linker may shrink a relative form or replace GOT/TLSDESC access with a direct form when the resolved symbol is local and in range.

9.3 medium Model Details

The `medium` model keeps code, GOT, and PLT within a signed 32-bit relative window but permits data objects anywhere in the 64-bit address space. Code uses direct signed 32-bit transfers. Data normally uses a GOT-resolved 64-bit address reached through a signed 32-bit GOT reference.

Its default relocations are `R_BEDROCK_BRDISP32S`, `R_BEDROCK_CALL32S`, `R_BEDROCK_GOTPCREL32S`, `R_BEDROCK_GOT64`, `R_BEDROCK_PLT32S`, and `R_BEDROCK_TLSDESC_GOTPCREL32S`. A linker may replace a GOT data reference with a direct relative reference when final placement puts the object in range.

9.4 high Model Details

The `high` model is a static high-address image. Internal code may use direct signed 32-bit transfers or image-base-relative addressing. Data uses a PC-relative GOT or an image-base-relative form when in range; the GOT and PLT are image-local or base-relative.

Its default relocations are `R_BEDROCK_BRDISP32S`, `R_BEDROCK_CALL32S`, `R_BEDROCK_GOTPCREL32S`, `R_BEDROCK_ABS64`, and `R_BEDROCK_DISP64`. Compact internal references are permitted when final placement proves the signed 32-bit range; full-width and base-relative sequences remain valid.

9.5 large Model Details

The `large` model permits arbitrary placement. Code uses full-width address materialization followed by an indirect transfer when necessary; data, GOT, and PLT addresses are full width.

Its default relocations are `R_BEDROCK_ABS64`, `R_BEDROCK_PCREL64`, `R_BEDROCK_GOTPCREL64`, `R_BEDROCK_PLT64`, and `R_BEDROCK_TLSDESC_GOTPCREL64`. Final placement may justify relaxing any full-width form to a shorter encoding.

10 DEBUGGING AND UNWIND ENCODING

The ELF ABI owns the architecture-wide DWARF register-number assignment used by every source language and unwind producer.

Table 10-1. Bedrock DWARF Register Numbers

Numbers	Registers	Status
0..15	`R0..R15`	assigned
16	`SP`	assigned
17	`PC`	assigned; CIE return-address register
18..19	`FLAGS..STATUS`	assigned
20..31	—	reserved
32..47	`F0..F15`	assigned
48..56	`CS..GS5`	assigned
57..58	`FFLAGS..FSTATUS`	assigned
59..63	—	reserved
64..95	`V0..V31`	extension; VECTOR extension present
96..111	`P0..P15`	extension; VECTOR extension present
112 and greater	—	reserved

An unwind description that saves a vector or predicate register describes its complete VLEN-dependent image. Predicate register locations contain the packed image with its exact architectural byte count.

11 ASSEMBLER CONTRACT

Default ABI: bedrock-elf

11.1 Strict Mode

- Pure ISA syntax is accepted.
- Ambiguous symbolic operands require an explicit relocation annotation.
- Noncanonical encodings are rejected.

11.2 Relaxation

- Branch and call payload widths may be selected automatically when the target is known.
- Address materialization may shrink from 64-bit to 32-bit, 16-bit, or 8-bit payloads when the relocation result fits.
- Overlong encodings are emitted only when explicitly requested.