BEDROCK ARCHITECTURE

Bedrock C ABI

C Language Binding and Calling Convention

Draft

CONTENTS

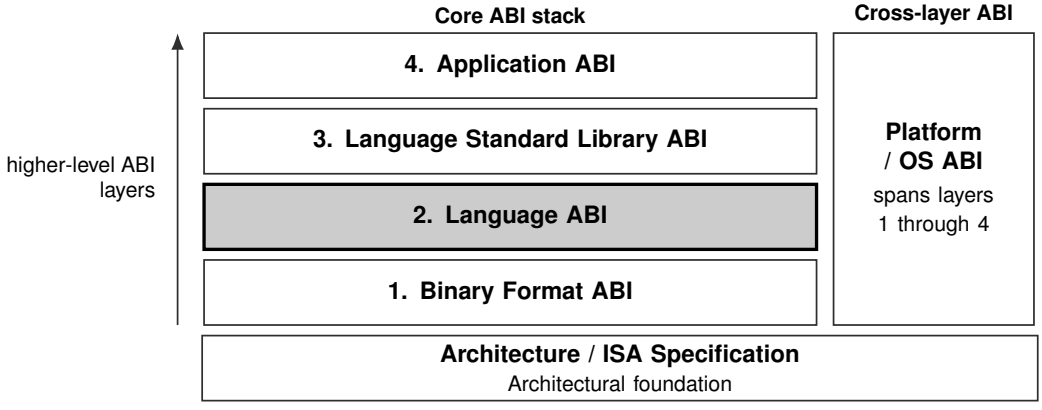
LIST OF TABLES

LIST OF FIGURES

1 SCOPE

This document defines the Bedrock C data model, register convention, calling convention, aggregate and variadic rules, compiler-runtime helper ABI, and C memory-model lowering. The Bedrock ELF ABI and ISA supply the referenced binary-format and architectural objects. Target-specific C extensions own their source syntax and semantics.

1.1 Contract Position



Current document: Bedrock C 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-c`

2 TERMINOLOGY

These terms are specific to the C binding. ELF and address-space terms keep the meanings defined by the Bedrock ELF ABI and the architecture manual.

object pointer A stable 64-bit pre-segment coordinate that designates an object or one-past an object according to the C abstract machine.

function pointer A stable 64-bit pre-segment coordinate naming a C entry point.

caller-saved register A register whose live incoming value the caller preserves across a call.

callee-saved register
A register whose incoming value must be preserved by the called function if the function uses it.

sret buffer Caller-allocated storage used to receive a return value that is too large or unsuitable for the normal return registers.

addressable by-value argument
A by-value aggregate argument that is passed through caller-owned stack storage because the callee may need an address for the object.

scalable value A target vector or predicate value whose complete object size is determined by the implementation-selected VLEN. Source-language spellings belong to the applicable compiler extension; this calling convention defines their binary interface.

3 DATA MODEL

Model: LP64

Byte Order: little endian

Internal SP: 8-byte aligned

Function Entry: 2-byte aligned

Before CALL: SP mod 16 = 8

Aggregate Maximum Alignment: 16 bytes

3.1 Scalar Types

Table 3-1. Reference C Data Model

Type	Bits	Align
`_Bool`	8	1
`char`	8	1
`signed char`	8	1
`unsigned char`	8	1
`short`	16	2
`int`	32	4
`long`	64	8
`long long`	64	8
`__int128`	128	16
`unsigned __int128`	128	16
`ordinary pointer`	64	8
`size_t`	64	8
`ptrdiff_t`	64	8
`wchar_t`	32	4
`char16_t`	16	2
`char32_t`	32	4
`float`	32	4
`double`	64	8
`long double`	64	8
`float _Complex`	64	4
`double _Complex`	128	8
`long double _Complex`	128	8

3.2 Scalar Type Semantics

The widths and natural alignments of scalar types are given in Table 3-1. The following rules define the type identities and representations that supplement those two values.

Integer types and aliases. Plain `char` is signed. `size_t` is an alias for `unsigned long`, and `ptrdiff_t` is an alias for `long`. `wchar_t` is a signed 32-bit integer; `char16_t` and `char32_t` are

unsigned 16-bit and 32-bit integers, respectively. In the absence of a separate enum extension, an enum has the representation, size, and alignment of `int`.

A stored `_Bool` has the integer value zero or one. False is encoded with every object-representation bit clear; true is encoded with bit 0 set and every other bit clear.

Ordinary pointers. An ordinary object or function pointer is 64 bits wide and 8-byte aligned. Its value is a pre-segment coordinate. Object and function pointers remain distinct C type categories even though their baseline representations have the same size and alignment. The null pointer has numeric address zero and an all-zero 64-bit object representation. Every ordinary-pointer result with numeric address zero is the null pointer.

128-bit integers. The types `__int128` and `unsigned __int128` are 16-byte, 16-byte-aligned integers. Their little-endian object representation places the low 64-bit word at the lower address. When the calling convention assigns one to a GENERAL-PAIR, the low word occupies the lower-numbered even register and the high word occupies the following odd register. A returned 128-bit integer therefore uses `R0` for the low word and `R1` for the high word. Its stack slot is 16-byte aligned.

An ordinary 128-bit load or store may use two 64-bit memory operations. The baseline C atomic object set contains the naturally aligned 1-, 2-, 4-, and 8-byte integer and pointer types.

Floating-point profile. The `bedrock-c` profile requires the base floating-point extension. Floating-point helpers may replace unavailable operations only when the `F0`–`F15` register calling convention remains available.

Long double. `long double` is a distinct C type with the IEEE-754 binary64 object representation, 8-byte size, and 8-byte alignment. It uses the floating-point argument class and returns in `F0`. It therefore has the same call representation as `double`. In a variadic call it occupies one 16-byte slot containing its 8-byte binary64 representation.

Complex types. A complex object contains its real component first and its imaginary component second, each using the corresponding real type's representation. Therefore `float _Complex` has size 8 and alignment 4, while `double _Complex` and `long double _Complex` have size 16 and alignment 8.

Scalable vector and predicate types. When the scalable-vector extension is present, let B be VLEN in bytes. A vector object contains exactly B bytes and has 16-byte alignment. A predicate object contains exactly $B/8$ packed-image bytes. A private vector spill slot has B bytes, 16-byte alignment, and B-byte stride. A private predicate spill slot has 16-byte alignment and $\mathrm{align\_up}(B/8, 16)$-byte stride. Predicate spill tail padding has unspecified contents. VLEN is stable until reset, so every object and frame in one execution context uses one size.

3.3 Aggregate Layout

Aggregate layout applies the scalar and pointer sizes and alignments defined above as follows:

- Struct fields are laid out in declaration order with natural alignment up to the aggregate maximum.
- A union's alignment is the greatest alignment required by any member. Its size is the greatest ABI size of any member, rounded up to a multiple of the union alignment.
- Arrays store elements contiguously with each element using the ABI size of its type.
- Aggregate object size is rounded up to the aggregate object's alignment.

Bit-fields. A bit-field allocation unit has the size and alignment of its declared base type. Within the little-endian unit, allocation begins at the low bit. A field remains within one allocation-unit boundary, and only consecutive fields whose base types are identical after typedef expansion may share a unit. A base-type change or insufficient remaining space closes the current unit and begins a new naturally aligned unit. A field width is at most the representation width of its base type.

An externally visible bit-field base type shall be `_Bool`, `signed int`, `unsigned int`, or a typedef of one of those types. A plain `int` bit-field has the value range and representation of `signed int`. The baseline ABI does not define an external calling or object-layout contract for bit-fields with any other base type.

An unnamed nonzero-width field consumes space normally. A zero-width field closes the current unit and advances layout to the next boundary for its base type while contributing zero storage bytes. Ordinary aggregate alignment, tail-padding, and size-rounding rules still apply after bit-field allocation.

Flexible arrays and extension layout facilities. A flexible array member contributes zero element-storage bytes to `sizeof` its containing structure. Its offset is aligned for the element type, and that element alignment participates in the structure's alignment calculation.

The baseline ABI uses the natural aggregate layout above. Packed aggregates, `#pragma pack`, empty structures, and zero-length arrays have no external calling or object-layout contract in this ABI.

4 REGISTER CONVENTION

This section defines which architectural state survives an ordinary C call. Argument and result locations are defined by the assignment procedure in Section 5; that procedure is their sole definition.

4.1 Register Preservation

The caller shall preserve any live value held in a volatile register before a call. A callee that modifies a nonvolatile register shall restore its exact incoming value before returning. Volatile registers carry explicitly assigned call results and caller-managed live values across the call.

Table 4-1. C Call Preservation Sets

State	Volatile across a call	Preserved by the callee
General registers	R0–R7	R8–R15
Floating-point registers	F0–F7	F8–F15
Vector registers	V0–V15	V16–V31
Predicate registers	P0–P7	P8–P15
Condition state	FLAGS	none
Segment state	GS1–GS5	CS, DS, SS, GS0

A callee may preserve floating-point registers with FPUSHP and FPOPP or with any sequence having the same observable effect. Each instruction operates on one canonical pair. The nonvolatile pairs are index 4 for F8:F9, index 5 for F10:F11, index 6 for F12:F13, and index 7 for F14:F15. When several pairs are saved, a normal prologue pushes them in increasing pair-index order and the matching epilogue pops them in decreasing pair-index order. A callee need only save the pair containing each nonvolatile register it modifies.

A callee that modifies a nonvolatile vector or predicate register shall restore that register's complete scalable image before returning. Saving only the currently active lanes, only significant predicate bits for one element size, or a fixed-width prefix represents a partial save; preservation requires the complete scalable image.

4.2 Stack and Control State

SP is the dedicated architectural stack pointer. A callee may move it while executing but shall restore the entry value before returning; Section 5 defines the additional alignment and frame rules. PC is the architectural program counter and is visible to C code only as a control-flow target. The call and return instructions define its transition across a call. Condition codes in FLAGS are volatile and may be clobbered by every callee.

4.3 Segment Context

An ordinary C call preserves the exact incoming images of CS, DS, SS, and GS0. The first three establish the code, ordinary-data, and stack contexts. GS0 establishes the TLS context and may use translated-window mode. GS1 through GS5 are caller-saved scratch segment registers; callers preserve any live incoming values they hold.

4.4 C Address-Space Context

Ordinary object and function pointers are stable 64-bit pre-segment coordinates. Every C ABI execution path maintains the following invariants:

- While a pointer is live, calls, returns, register or memory preservation, context transitions, and ABI boundaries preserve its 64-bit coordinate.
- Every ABI-permitted segment path that dereferences a valid object pointer selects the same byte of the same C object for a given pointer value. This includes a stack object's address after it escapes to an ordinary data path, and a materialized TLS object's address.

- Pointer equality, subtraction, and in-object arithmetic have the same C result on every ABI-permitted path. A one-past pointer participates in those operations and is an arithmetic boundary sentinel. Storage accesses use coordinates within the object.
- Function pointers, call entries, return continuations, ELF symbols and relocations, dynamic linkage, stack pointers, runtime-provided pointers, TLS materialization, signal and nonlocal-jump contexts, unwind records, and debugger-visible coordinates use this pre-segment coordinate system.
- Segment and page-table context transitions preserve these invariants as one architectural transition from the program's perspective.
- Every C object and its one-past coordinate must be representable as a nonzero 64-bit address, including its one-past coordinate. The greatest representable one-past coordinate is $2^{64} - 1$.

The 64-bit all-zero null-pointer representation designates the null pointer. Every C object and function occupies a nonzero address.

4.5 Status Registers

Target-specific intrinsics and runtime code provide C access to `STATUS`. Low-level access to `FSTATUS` and `FFLAGS` uses `<bedrockfpuintrin.h>`. `FSTATUS` is callee-saved: a function returns with the incoming rounding mode, floating-point exception-condition enables, and other control fields. A documented floating-point-environment interface instead specifies its outgoing environment explicitly. `FFLAGS` is cumulative floating-point status and is caller-clobbered; floating-point operations in a callee may set accrued floating-point exception flags. An interface that requires a saved environment uses its floating-point environment runtime contract explicitly.

5 CALLING CONVENTION

This section is normative. A caller and callee conform when they independently apply the classification and assignment procedure below and therefore agree on every argument location, return location, and stack boundary. Register lists are summaries; the assignment procedure remains authoritative.

5.1 Stack and Calls

The stack grows downward. An architectural `CALL` pushes an 8-byte return address at `[SP]`, and callers align `SP` before the call so callee entry observes 16-byte alignment. Between ABI call boundaries, a function may adjust `SP` while keeping it aligned to at least 8 bytes. Immediately before `CALL`, the stack pointer satisfies SP modulo 16 equals 8 before return address push. After `CALL` pushes the return address, it satisfies SP modulo 16 equals 0 at callee entry.

Each object observes its declared type alignment. In particular, relaxing the internal stack-pointer invariant preserves the 16-byte alignment of `__int128` objects or other objects whose declared alignment is 16 bytes. Both the red-zone size and fixed outgoing-argument-area size are zero. A callee that dynamically realigns the stack restores the incoming stack pointer before returning.

The first stack argument begins at `[SP+16]` at callee entry. The eight bytes at `[SP+8]` are alignment padding. Argument values begin at `[SP+16]`. Each subsequent stack argument occupies the next 16-byte slot.

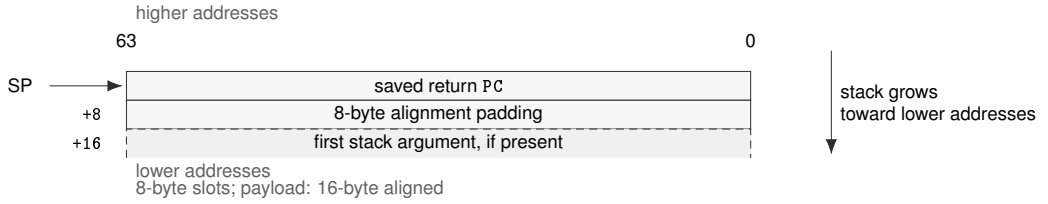


Figure 5-1. Near-call stack frame at callee entry

5.2 Frame Pointer

A function that maintains a frame pointer uses callee-saved `R15`. Other functions retain `R15` as an ordinary callee-saved register.

5.3 C Unwind Frame

The Bedrock ELF ABI owns the language-independent DWARF register-number assignment. C unwind descriptions use that assignment with the following calling-convention frame rule. At call entry, the canonical frame address is entry `SP+8`; saved `PC` is at CFA minus 8 and `CS` has a same-value rule.

5.4 Argument Classification

Classification occurs after the C language has determined the effective type of the argument. A named argument to a prototyped function is placed in one of seven ABI classes:

GENERAL	Integer types up to 64 bits wide and ordinary object or function pointers. One general-register location is required.
GENERAL-PAIR	Signed and unsigned `__int128`. Two consecutive general registers beginning at an even-numbered register are required.
FLOAT	`float`, `double`, and `long double`. One floating-point register is required.
FLOAT-PAIR	`float _Complex`, `double _Complex`, and `long double _Complex`. Two consecutive floating-point registers are required with unit register alignment.
VECTOR	A direct scalable vector value. One vector register is required.
PREDICATE	A direct scalable predicate value. One predicate register is required.
INDIRECT	Every structure or union argument, regardless of size or member composition. The caller creates a 16-byte-aligned by-value copy and passes the address of that copy as one GENERAL value.

A compound scalable value is INDIRECT. Exhaustion of a scalable value's direct register class also converts it to INDIRECT: the caller creates an object with the layout above and assigns its pointer through the GENERAL procedure. Each scalable value travels wholly in its register or through an indirect copy.

Signed GENERAL values narrower than 64 bits are sign-extended when assigned to a register. Unsigned values and `_Bool` are zero-extended. Plain `char` follows its signed ABI type. A stack slot contains the effective type's object representation at its low address; bytes after that representation are unspecified.

5.5 Assignment Procedure

The caller shall apply the following procedure. The callee may rely on the result solely from assigned registers and value bytes.

1. If the result uses an sret buffer, assign its address to `R0` and initialize the general-register cursor to 1. Otherwise initialize it to 0. Initialize the floating-point cursor to 0, the vector cursor to 0, and the predicate cursor to 0.
2. Visit arguments in source order. GENERAL and INDIRECT arguments consume the register named by the current general cursor and then advance that cursor. FLOAT arguments analogously consume `F0` through `F7` using the independent floating-point cursor. VECTOR arguments consume `V0` through `V7` using the independent vector cursor, and PREDICATE arguments consume `P0` through `P3` using the independent predicate cursor.
3. For a FLOAT-PAIR argument, use the current floating-point register for the real component and the next register for the imaginary component, then advance the cursor by two with unit register alignment. If fewer than two registers remain, place the complete complex value in one stack slot and mark the floating-point class exhausted.
4. For a GENERAL-PAIR argument, round the general cursor upward to an even register number. If that register and its successor both exist, place the low 64 bits in the even register, the high 64 bits in the odd register, and advance the cursor past the pair. A skipped odd register remains unused.
5. If the remaining registers of a class have insufficient capacity for an argument, place the complete argument in one stack slot and mark that register class exhausted. Later arguments of that class also use stack slots, while register holes remain unassigned. Each argument occupies one complete register location or one complete stack slot. For VECTOR or PREDICATE, instead convert the complete argument to INDIRECT as specified above and assign that pointer through GENERAL; exhausting a scalable register class leaves GENERAL independently available.
6. Unnamed variadic arguments and every argument of an unprototyped call are forced to stack slots and leave register cursors unchanged. Apply the default argument promotions before assigning those slots. Every scalable argument in either case is first converted to INDIRECT. Its GENERAL pointer is then subject to this forced-stack rule, so the variadic slot contains the pointer exclusively.
7. Assign stack slots to stack-bound arguments in source order beginning at `[SP+16]`. Consequently, a caller that constructs the outgoing area by stores or pushes does so from the

rightmost stack-bound argument toward the leftmost. The reserved register-home-slot count is zero.

The general, floating-point, vector, and predicate resources exhaust independently.

5.6 C Return Values

Scalar results use their natural register class. A `__int128` result places its low word in `R0` and high word in `R1`. A structure or union of at most 16 bytes is returned as raw object bytes: increasing object addresses map first to `R0` and then to `R1`. Bytes beyond the object size and padding bytes with C-unspecified values are unspecified.

A complex result places its real component in `F0` and its imaginary component in `F1`.

A single direct vector result uses `V0`; a single direct predicate result uses `P0`. A compound scalable result or any result containing multiple scalable values uses the ordinary indirect-result procedure, even if its fixed fields would otherwise fit in `R0:R1`.

For a larger aggregate, the caller allocates a 16-byte-aligned result buffer and passes its address in `R0` before ordinary argument assignment. The callee stores the result in that buffer and returns the same address in `R0`.

Table 5-1. C Return Register Quick Reference

Result kinds	Register rule
`bool, i8, u8, i16, u16, i32, u32, i64, u64, pointer, function_pointer`	`R0`
`i128, u128`	`R1:R0`
`f32, f64, long_double`	`F0`
`complex_f32, complex_f64, complex_long_double`	`real component in F0; imaginary component in F1`
`vector`	`V0`
`predicate`	`P0`
`aggregate`	`up to 16 bytes in R1:R0; larger values use sret`
`compound_scalable`	`sret pointer in R0; result pointer in R0`

5.7 Aggregate Passing

For every by-value aggregate argument, the caller allocates distinct storage, aligns its start to 16 bytes, initializes it with the argument value, and keeps it alive until the call returns. The GENERAL procedure assigns the address exclusively. If GENERAL registers are exhausted, the stack argument slot contains that address exclusively.

The copy gives the callee an addressable object with the declared aggregate type. Modifications to the private copy remain isolated from the caller's source object. Baseline aggregates containing bit-fields and mixed integer/floating aggregates use the same INDIRECT rule; their member composition preserves the INDIRECT classification. An extension aggregate with packed or unaligned members is outside this ABI contract.

5.8 Variadic Calls

Fixed named arguments use the ordinary assignment procedure. Every unnamed argument is first subject to the C default argument promotions and is then placed in a 16-byte stack slot. In particular, `float` becomes `double`, and `_Bool`, `char`, `short`, and their signed or unsigned variants promote to `int` when `int` can represent all values of the source type. `long double` remains the Bedrock binary64 `long double` type. An aggregate slot contains the address of the caller-owned by-value copy described above.

The baseline `va_list` representation is one 64-bit pointer to the next unnamed argument slot. `va_start` initializes it to the first unnamed slot after any stack-assigned named arguments. `va_arg` reads the promoted representation from the low bytes of the current slot, or follows the copy address for an aggregate, and then advances the pointer by 16 bytes. `va_copy` copies the pointer value and `va_end` has a zero-instruction machine effect.

An unprototyped call applies the same promotions and places every argument on the stack. As required by C, the callee type must be compatible with the promoted types.

6 FREESTANDING C BINDING

This section defines the compiler-runtime and ELF linkage obligations of the freestanding C profile. It covers compiler-generated helper calls, C symbols, and ELF linkage.

6.1 Compiler Runtime Helpers

The compiler lowers addition, subtraction, negation, bitwise operations, comparisons, truncation, and extension of 128-bit integers inline. It may call one of the helpers below for 128-bit multiplication, division, remainder, and shifts; conversion between a 128-bit integer and a supported floating-point type; or complex multiplication and division.

Every compiler-runtime helper is an ordinary `bedrock-c` function. Its arguments and results therefore follow Section 5, including the `R1:R0` return convention for `__int128`.

The tables below are the complete baseline helper set. A compiler emits implicit references from this set or from a target extension that defines the symbol and its ABI. A runtime may export additional symbols as explicit interfaces.

In the signature column, `i128` means `__int128`, `u128` means `unsigned __int128`, `c32` means `float _Complex`, and `c64` means `double _Complex`. The signatures use ordinary C parameter and result classification; a shift count has type `int`.

Table 6-1. `__int128` Arithmetic and Shift Helpers

Operation	Helper	Signature
Multiply	`__multi3`	`i128(i128, i128)`
Signed divide	`__divti3`	`i128(i128, i128)`
Unsigned divide	`__udivti3`	`u128(u128, u128)`
Signed remainder	`__modti3`	`i128(i128, i128)`
Unsigned remainder	`__umodti3`	`u128(u128, u128)`
Shift left	`__ashlti3`	`i128(i128, int)`
Arithmetic shift right	`__ashrti3`	`i128(i128, int)`
Logical shift right	`__lshrti3`	`u128(u128, int)`

Table 6-2. `__int128` and Floating-Point Conversion Helpers

Conversion	Helper	Signature
Signed to binary32	`__floattisf`	`float(i128)`
Signed to binary64	`__floattidf`	`double(i128)`
Unsigned to binary32	`__floatuntisf`	`float(u128)`
Unsigned to binary64	`__floatuntidf`	`double(u128)`
Binary32 to signed	`__fixsfti`	`i128(float)`
Binary64 to signed	`__fixdfti`	`i128(double)`
Binary32 to unsigned	`__fixunssfti`	`u128(float)`
Binary64 to unsigned	`__fixunsdfti`	`u128(double)`

Table 6-3. Complex Arithmetic Helpers

Operation	Helper	Signature
Binary32 multiply	`__mulsc3`	`c32(float, float, float, float)`
Binary32 divide	`__divsc3`	`c32(float, float, float, float)`
Binary64 multiply	`__muldc3`	`c64(double, double, double, double)`
Binary64 divide	`__divdc3`	`c64(double, double, double, double)`

The four scalar arguments to a complex helper are the real and imaginary parts of its two operands, in operand order. Bedrock `long double` has the binary64 representation and call classification, so conversions involving `long double` use the corresponding `df` helper and `long double _Complex` multiplication or division uses the corresponding `dc3` helper.

6.2 Pointers and External Objects

A C function pointer is 64 bits wide and its value is a pre-segment near-call coordinate. Code and data pointers have the same size and alignment, while their distinct C type categories preserve separate code and data address spaces.

A common symbol has the ABI alignment of its declared type. An external object retains its declared alignment up to the 16-byte aggregate maximum. An explicit extension owns stronger placement requirements.

6.3 Calls, Relocations, and Relaxation

The relocation records the linkage path selected for a C call:

Table 6-4. C Call Relocation Quick Reference

Call form	Relocation
Direct C call	`R_BEDROCK_CALL32S`
External call through the PLT	`R_BEDROCK_PLT32S`

The assembler and linker may relax an instruction sequence, but relaxation must preserve the C call interface, symbol identity, and observable control transfer defined by the original relocation. The relaxed sequence remains part of the same C calling convention.

6.4 TLS Relationship

The Bedrock ELF ABI defines TLS models, symbols, relocations, and the `GS0`-relative materialization protocols consumed by C code.

7 C MEMORY MODEL

This section defines how the baseline C memory model maps to Bedrock memory operations and ordering instructions.

The ISA memory model owns access atomicity, alignment, instruction-order selectors, and the global sequentially consistent order. This C ABI owns atomic-object eligibility and the lowering of C operations into those ISA contracts.

7.1 Atomic Object Eligibility

The native lock-free set consists of integer objects and ordinary 64-bit pointer objects whose size is 1, 2, 4, or 8 bytes and whose address satisfies the natural alignment for that size. An unaligned atomic object requires the target constraint diagnostic below. Accordingly, `__atomic_always_lock_free` returns true only for those naturally aligned sizes.

The baseline accepts an atomic object only when it belongs to that native lock-free set. Any other atomic object—including a floating-point, aggregate, 128-bit integer, unsupported-width, or insufficiently aligned object—is a target constraint violation.

7.2 Native Atomic Primitives

Once the object satisfies the width and alignment requirements above, the compiler uses the following primitive lowering. The ordering sequences are defined separately below.

Table 7-1. Native Atomic Primitive Quick Reference

C operation	Bedrock lowering
`atomic_load`	`aligned MOV access plus the order sequence`
`atomic_store`	`aligned MOV access plus the order sequence`
`atomic_compare_exchange`	`CMPXCHG`
`atomic_exchange`	`loop using CMPXCHG`

7.3 C Atomic Memory Order Mapping

Table 7-2. C Atomic Memory Order Mapping

C order	Instruction	Load	Store	Fence
relaxed	relaxed	load	store	zero instructions
consume	acquire	load; AFENCE	—	AFENCE
acquire	acquire	load; AFENCE	—	AFENCE
release	release	—	AFENCE; store	AFENCE
acq_rel	acqrel	—	—	AFENCE
seq_cst	seqcst	AFENCE; load; AFENCE	AFENCE; store; AFENCE	AFENCE

For C compare-exchange, the compiler selects the join of the success and failure orders through the order lattice, independently of their numeric encodings. Consume is treated as acquire, acquire joined with release is acquire-release, and sequential consistency dominates every other order. A failure performs a read-only operation. Release-sequence effects arise from successful writes, while a failed `CMPXCHG.SEQCST` still participates in the global SC order as an SC load.

The relaxed C thread fence emits a zero-instruction sequence. Consume, acquire, release, acquire-release, and sequentially consistent thread fences emit `AFENCE`.

7.4 Volatile, Device, and Executable Memory

Volatile accesses preserve the observable access count and order. C atomic operations and fences provide the inter-thread ordering defined above. The C target interface owns the executable-byte publication operation. A relocation agent completes that operation before transferring control to patched instructions.

7.5 C Fetch RMW Lowering

Table 7-3. C Fetch RMW Lowering

C operation	Bedrock lowering
atomic_fetch_add	FETCHADD
atomic_fetch_sub	FETCHSUB
atomic_fetch_and	FETCHAND
atomic_fetch_or	FETCHOR
atomic_fetch_xor	FETCHXOR

8 CALLING CONVENTION EXAMPLES

The examples apply the normative assignment procedure above.

8.1 Independent Register Classes and Pair Alignment

```
struct pair { uint64_t lo, hi; };
uint64_t mixed(uint64_t count, double scale,
  unsigned __int128 wide, struct pair pair, float bias);
```

The general cursor begins at 0 and the floating-point cursor begins at 0. `count` consumes `R0`; `scale` independently consumes `F0`. The GENERAL-PAIR argument rounds the general cursor from 1 to 2, so `wide` uses `R3:R2` and `R1` remains unused. The address of the caller-owned `pair` copy uses `R4`, and `bias` uses `F1`. The stack argument area has size zero.

8.2 Independent Scalable Register Classes

Assume `vec` and `pred` denote direct scalable vector and predicate extension types. For `vec blend(vec left, pred mask, uint64_t count, vec right, pred select, double scale)`, `left` and `right` use `V0` and `V1`; `mask` and `select` independently use `P0` and `P1`; `count` uses `R0`; and `scale` uses `F0`. The direct vector result uses `V0`. Each class advances its own cursor exclusively.

8.3 sret Changes Ordinary Argument Assignment

```
struct triple { uint64_t x, y, z; };
struct triple build(uint64_t tag, signed __int128 wide,
  double factor);
```

Because the result is 24 bytes, its buffer address occupies `R0` and the general cursor begins at 1. Thus `tag` uses `R1`, `wide` uses `R3:R2`, and `factor` uses `F0`. The callee stores the result through the entry value of `R0` and returns the same address in `R0`.

8.4 Pair Exhaustion of the General Class

```
void pressure(uint64_t a0, uint64_t a1, uint64_t a2,
  uint64_t a3, uint64_t a4, uint64_t a5, uint64_t a6,
  unsigned __int128 wide, uint64_t tail);
```

Arguments `a0` through `a6` use `R0` through `R6`. The next even pair would begin past `R7`, so `wide` is placed completely at `[SP+16]` and the GENERAL class becomes exhausted. `tail` therefore uses `[SP+32]`, while `R7` remains unassigned.

8.5 Variadic Promotions and Slot Traversal

```
int trace(const char *format, ...);
trace("example", (signed char)-1, (float)1.5, pair);
```

The named `format` argument uses `R0`. The signed character is promoted to `int` and stored at `[SP+16]`; the `float` is promoted to `double` and stored at `[SP+32]`. The caller creates the

aggregate copy and stores its address at [SP+48]. A va_list initialized by va_start visits those three slots in that order by adding 16 after each access.

8.6 Assembly Conventions

The assembly examples illustrate the normative rules. They present the directives, comments, and labels that determine the shown control flow. The scalar leaf example presents its one-instruction counted loop with REP; the counter supplies N directly and zero annuls the body. If an implementation sequence conflicts with a normative rule, the rule controls.

The declarations below state the ABI interfaces used by the examples.

8.7 Scalar Leaf Function

```
int sum(int *p, int n);
```

On entry, p is in R0 and the sign-extended value of n is in R1. The function uses only volatile registers, preserves the entry SP, and returns the integer sum in R0.

```
sum:
  mov.q  r0, r2
  clr.q  r0
  maxs.q r0, r1
  rep  r1, add.l [r2++], r0
  ret
```

Figure 8-1. Scalar leaf function

8.8 Non-Leaf Preservation and Call Alignment

```
int mf_pipeline(int *tmp, int *dst, int *src, int n,
  int ca, int cb, int cc);
```

The seven arguments occupy R0 through R6 in source order. The function keeps n, dst, tmp, and the first intermediate result live in nonvolatile R8 through R11. It therefore saves and restores both canonical register pairs.

```
mf_pipeline:
  pushp  4
  pushp  5
  sub.q  8, sp
  mov.q  r3, r8
  mov.q  r1, r9
  mov.q  r0, r10
  mov.q  r2, r1
  mov.q  r8, r2
  mov.q  r4, r3
  mov.q  r5, r4
  mov.q  r6, r5
  call   mf_fir_three
  mov.q  r0, r11
  mov.q  r9, r0
  mov.q  r10, r1
  mov.q  r8, r2
  call   mf_scale_store
  add.l  r11, r0
  mov.l r0, r2
  mov.q  r9, r0
  mov.q  r8, r1
  add.q  8, sp
  popp  5
  popp  4
  jmp   mf_bias_sum
```

Figure 8-2. Non-leaf function with a tail transfer

Each `PUSHP` saves 16 bytes, so the pair saves leave the entry alignment unchanged. The additional 8-byte subtraction establishes `SP mod 16 = 8` before each `CALL`; the pushed return address therefore gives the callee a 16-byte-aligned entry stack. The epilogue removes the padding and restores `R10:R11` before `R8:R9`. The final `JMP` is a legal tail transfer because the frame and nonvolatile state have already been restored, the outgoing arguments have the ordinary ABI layout, and the target's `R0` result is also the result required by `mf_pipeline`.