Artefact Design & Testing; Oops!Mate

Name: Swoyam Pokharel
Student Number: 2431342
Supervisor: Prakriti Regmi
Reader: Siman Giri

Submitted On: December 24, 2025

Contents

1 · Types Crate

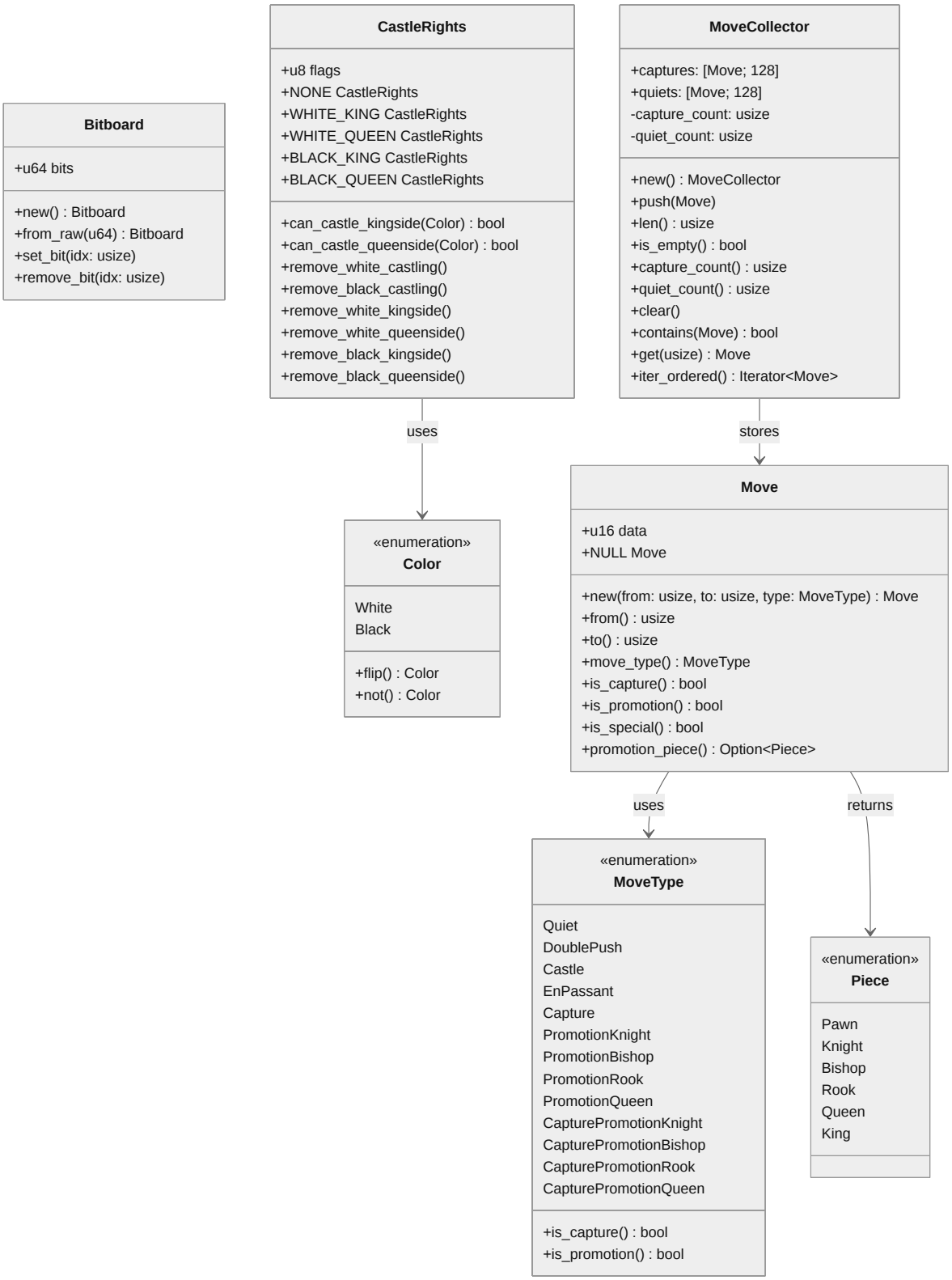


Figure 1: Class Diagram For The `types` crate

1.1 · Introduction

1.1.1 · Purpose

This section specifies the software requirements for the `types` crate of the chess engine. The `types` crate provides fundamental data structures and type definitions used throughout the engine.

1.1.2 · Scope

The `types` crate serves as the foundational layer, defining core chess concepts including bitboard representation, move encoding, piece types, and move collection structures optimized for performance.

1.2 · Functional Requirements

ID	Description	Type	Status
TYPES-F-001	Bitboard must represent 64 squares as u64 with bit manipulation operations (set, remove, OR, AND)	Functional	Complete
TYPES-F-002	Color enumeration must represent White and Black with toggle operations (NOT operator and flip method)	Functional	Complete
TYPES-F-003	Piece enumeration must define all six piece types (Pawn, Knight, Bishop, Rook, Queen, King) mapped to internal representation	Functional	Complete
TYPES-F-004	CastleRights must store 4-bit flags for White/Black kingside/queenside with query and removal methods	Functional	Complete
TYPES-F-005	Move must encode source square (6 bits), destination square (6 bits), and move type (4 bits) in 16-bit value	Functional	Complete
TYPES-F-006	Move must support extraction of from square, to square, and move type through bit masking	Functional	Complete
TYPES-F-007	MoveType enumeration must define all move types: Quiet, DoublePush, Castle, EnPassant, Capture, and 8 promotion variants	Functional	Complete
TYPES-F-008	Move must identify captures and promotions through bit flag checking (bit 2 for capture, bit 3 for promotion)	Functional	Complete
TYPES-F-009	Move must identify special moves (EnPassant, Castle, DoublePush) and extract promotion piece type	Functional	Complete
TYPES-F-010	Move must format to UCI string notation (e.g., “e2e4”, “e7e8q”) via Display trait	Functional	Complete
TYPES-F-011	MoveCollector must maintain separate arrays for captures (128 slots) and quiet moves (128 slots)	Functional	Complete

TYPES-F-012	MoveCollector must automatically route pushed moves to captures or quiets array based on move type	Functional	Complete
TYPES-F-013	MoveCollector must provide count queries (total, captures, quiets) and emptiness check	Functional	Complete
TYPES-F-014	MoveCollector must support move lookup by index (captures first, then quiets) and containment checking	Functional	Complete
TYPES-F-015	MoveCollector must provide ordered iteration (all captures before any quiets) for move ordering optimization	Functional	Complete
TYPES-F-016	System must convert square indices (0-63) to algebraic notation strings (e.g., 0 → “a1”, 63 → “h8”)	Functional	Complete

1.3 · Non-Functional Requirements

ID	Description	Type	Status
TYPES-NF-001	All critical operations (move encoding/decoding, bit operations) must be inlined (inline(always)) for zero-overhead abstraction	Non-Functional	Complete
TYPES-NF-002	Memory layout must be optimal: Bitboard (8B), Move (2B), Color/Piece/MoveType (1B each), CastleRights (1B)	Non-Functional	Complete
TYPES-NF-003	MoveCollector must use MaybeUninit for uninitialized memory to avoid zeroing overhead on allocation	Non-Functional	Complete
TYPES-NF-004	All enums must use repr(u8) for guaranteed layout and FFI compatibility; Bitboard uses repr(transparent)	Non-Functional	Complete

1.4 · Dependencies

- `std::mem::MaybeUninit` - Uninitialized memory management for performance
- `std::ops` - Operator overloading traits (BitOr, BitAnd, Not, Index)
- `crate::others::Piece` - Internal piece type dependency

1.5 · Test Plan: Types Crate

This test plan covers unit and integration testing for all types defined in the `types` crate, including Bitboard, Color, Piece, CastleRights, Move, MoveType, and MoveCollector.

1.5.1 · Test Cases

TC-001: Bitboard Operations

Test	Input	Expected Output
Empty bitboard	`Bitboard::new()`	`0x0`
Set bit 0	`bb.set_bit(0)`	`0x1`
Set bit 63	`bb.set_bit(63)`	`0x8000000000000000`
Remove bit	`bb.set_bit(5); bb.remove_bit(5)`	`0x0`
OR operation	`Bitboard(0x5) | Bitboard(0x3)`	`Bitboard(0x7)`
AND operation	`Bitboard(0x5) & Bitboard(0x3)`	`Bitboard(0x1)`

TC-002: Color Operations

Test	Input	Expected Output
NOT operator	`!Color::White`	`Color::Black`
NOT operator	`!Color::Black`	`Color::White`
Flip white	`Color::White.flip()`	`Color::Black`
Flip black	`Color::Black.flip()`	`Color::White`

TC-003: Move Encoding

Test	Input	Expected Output
Quiet move	`Move::new(12, 20, MoveType::Quiet)`	`from=12, to=20, type=Quiet`
Capture move	`Move::new(35, 44, MoveType::Capture)`	`from=35, to=44, is_capture=true`
Promotion	`Move::new(48, 56, MoveType::PromotionQueen)`	`is_promotion=true, piece=Queen`
En passant	`Move::new(33, 42, MoveType::EnPassant)`	`is_special=true`
Castle	`Move::new(4, 6, MoveType::Castle)`	`is_special=true`

TC-004: Move Type Detection

Test	Input	Expected Output
Is capture check	`MoveType::Capture.is_capture()`	`true`
Is capture check	`MoveType::Quiet.is_capture()`	`false`
Is promotion check	`MoveType::PromotionQueen.is_promotion()`	`true`
Capture promotion	`MoveType::CapturePromotionKnight.is_capture()`	`true`
Capture promotion	`MoveType::CapturePromotionKnight.is_promotion()`	`true`

TC-005: UCI String Formatting

Test	Input	Expected Output
Quiet move	`Move::new(12, 20, MoveType::Quiet)`	`"e2e3"`
Long move	`Move::new(0, 63, MoveType::Quiet)`	`"a1h8"`
Promotion	`Move::new(48, 56, MoveType::PromotionQueen)`	`"a7a8q"`
Knight promo	`Move::new(51, 59, MoveType::PromotionKnight)`	`"d7d8n"`
Square to string	`square_to_string(0)`	`"a1"`
Square to string	`square_to_string(63)`	`"h8"`

TC-006: MoveCollector Basic Operations

Test	Input	Expected Output
New collector	`MoveCollector::new()`	`len=0, is_empty=true`
Push quiet	`mc.push(quiet_move)`	`quiet_count=1, capture_count=0`
Push capture	`mc.push(capture_move)`	`capture_count=1`
Mixed moves	`push 3 captures, 5 quiets`	`len=8, capture_count=3`
Clear	`mc.clear()`	`len=0, is_empty=true`

TC-007: MoveCollector Indexing and Iteration

Test	Input	Expected Output
Get capture	`mc.get(0)` with captures	First capture move
Get quiet	`mc.get(capture_count)`	First quiet move
Index operator	`mc[0]`	First capture if exists
Contains check	`mc.contains(move)`	`true` if move exists
Ordered iteration	`mc.iter_ordered()`	Captures first, then quiets

1.5.2 · Environment

- Rust toolchain: stable
- Test framework: Built-in `cargo test`

2 · Raw Crate (Move Generation Crate; Raw Attack Tables)

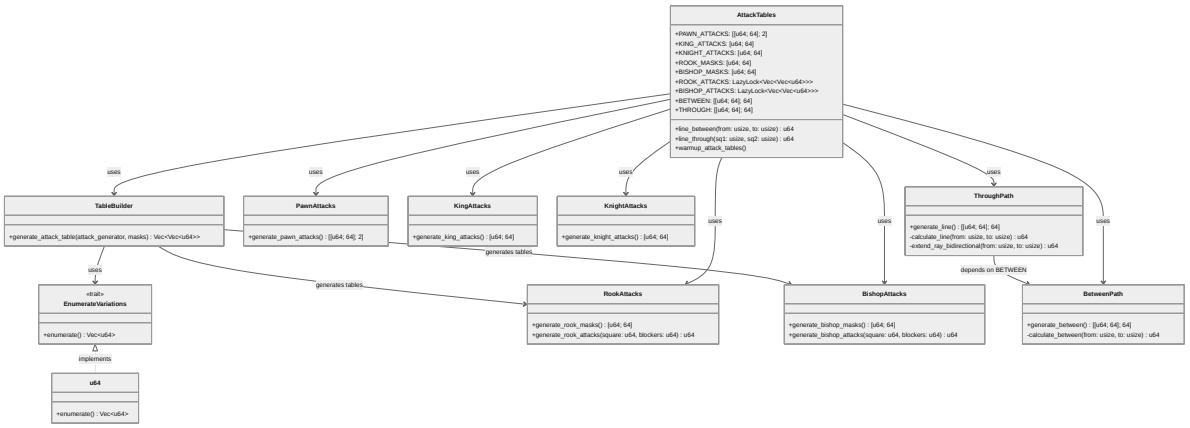


Figure 2: Class Diagram For The `raw` crate

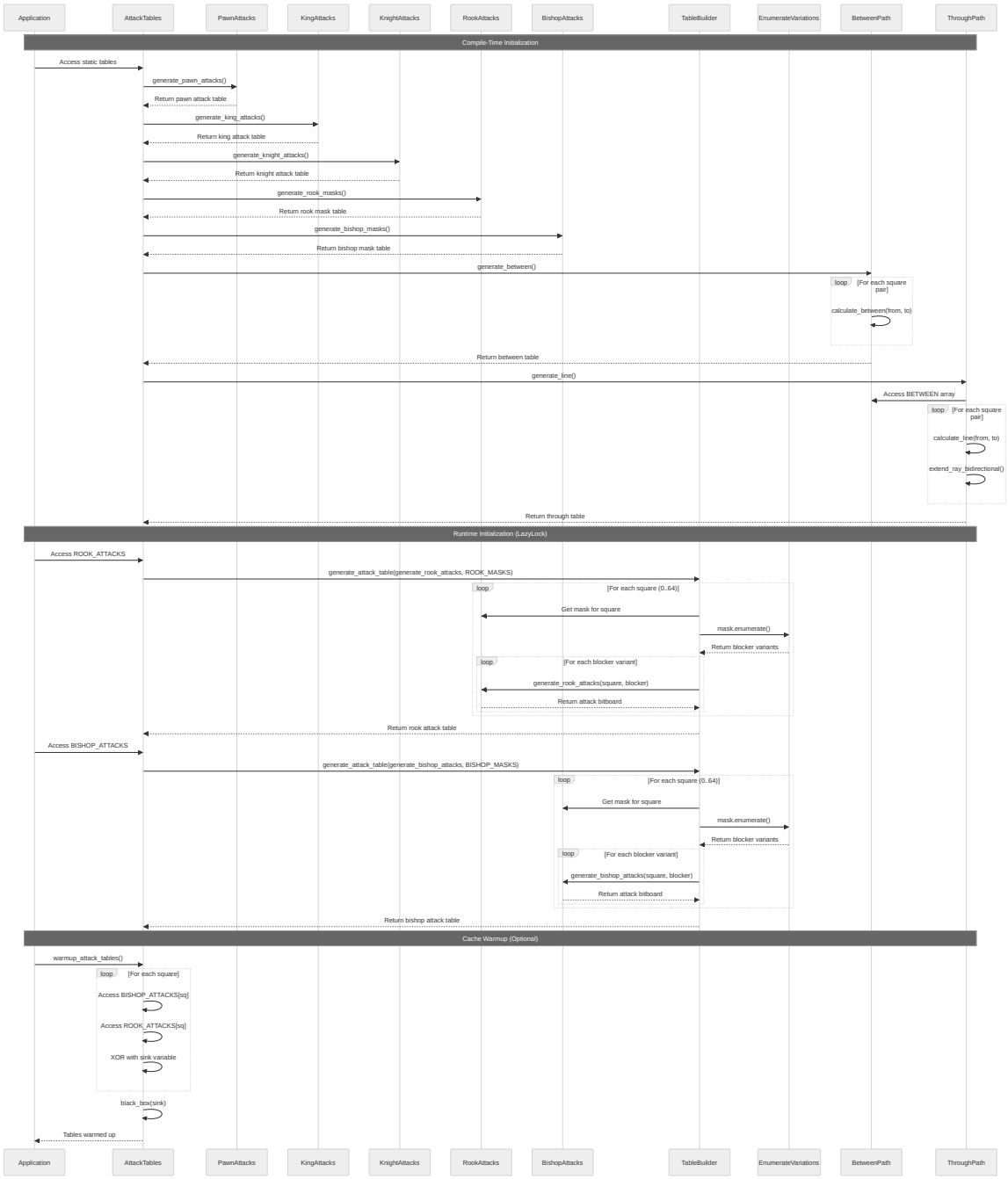


Figure 3: Sequence Diagram For The `raw` crate

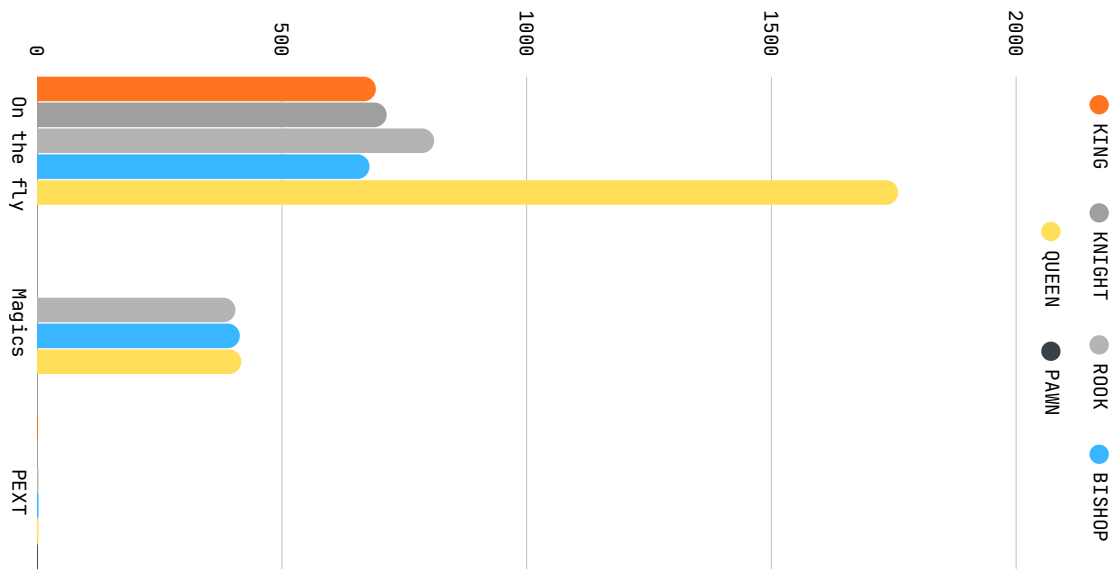


Figure 4: Comparison of Performance Between Different Move Generation Methods

label	king	knight	rook	bishop	queen	pawn
on the fly	692 n/s	714 n/s	811 n/s	679 n/s	1759 n/s	—
magics	—	—	271.59 n/s	277.13 n/s	308.78 n/s	—
pext	0.47 n/s	0.47 n/s	3.35 n/s	3.40 n/s	2.96 n/s	0.23 n/s

2.1 · Introduction

2.1.1 · Purpose

This section specifies the software requirements for the `raw` crate of the chess engine. The `raw` crate provides pre-computed attack tables and movement patterns for all piece types using bitboard representation.

2.1.2 · Scope

The `raw` crate serves as the lookup layer, generating and storing attack patterns for pawns, knights, bishops, rooks, kings, and queens. It provides efficient move generation through pre-computed tables and PEXT-based indexing for sliding pieces.

2.2 · Functional Requirements

ID	Description	Type	Status
RAW-F-001	System must generate pawn attack tables for both colors across all 64 squares at compile time	Functional	Complete
RAW-F-002	System must generate king attack tables for all 64 squares with 8-directional movement at compile time	Functional	Complete
RAW-F-003	System must generate knight attack tables for all 64 squares with L-shaped movement at compile time	Functional	Complete

RAW-F-004	System must generate rook mask tables excluding edge squares for PEXT indexing at compile time	Functional	Complete
RAW-F-005	System must generate bishop mask tables excluding edge squares for PEXT indexing at compile time	Functional	Complete
RAW-F-006	System must generate rook attack tables for all blocker configurations using PEXT indexing at runtime via LazyLock	Functional	Complete
RAW-F-007	System must generate bishop attack tables for all blocker configurations using PEXT indexing at runtime via LazyLock	Functional	Complete
RAW-F-008	System must provide line_between function returning exclusive squares between two squares on same rank, file, or diagonal	Functional	Complete
RAW-F-009	System must provide line_through function returning all squares on rank, file, or diagonal containing two given squares	Functional	Complete
RAW-F-010	EnumerateVariations trait must generate all possible variations of a binary pattern using carry-rippler algorithm	Functional	Complete
RAW-F-011	Attack table generator must accept attack generation function and mask array to produce PEXT-indexed attack tables	Functional	Complete
RAW-F-012	Rook attack generation must cast rays in 4 directions until blockers or board edges	Functional	Complete
RAW-F-013	Bishop attack generation must cast rays in 4 diagonal directions until blockers or board edges	Functional	Complete
RAW-F-014	System must provide warmup_attack_tables function to force initialization and CPU cache loading of all attack tables	Functional	Complete
RAW-F-015	Between table must return 0 for squares not on same rank, file, or diagonal	Functional	Complete

2.3 · Non-Functional Requirements

ID	Description	Type	Status
TABLES-NF-001	All table lookup functions must be inlined for zero-overhead access	Non-Functional	Complete
TABLES-NF-002	Fixed tables for pawns, knights, kings, and masks must be generated at compile time to reduce startup cost	Non-Functional	Complete
TABLES-NF-003	Sliding piece attack tables must use LazyLock for deferred initialization to reduce binary size	Non-Functional	Complete

TABLES-NF-004	Attack table generation must use PEXT instruction for efficient blocker configuration indexing	Non-Functional	Complete
TABLES-NF-005	Warmup function must access all table entries to ensure CPU cache loading before search begins	Non-Functional	Complete
TABLES-NF-006	All const generation functions must be const fn to enable compile-time evaluation	Non-Functional	Complete

2.4 · Dependencies

- `std::sync::LazyLock` - Deferred initialization for large attack tables
- `std::arch::x86_64::_pext_u64` - PEXT instruction for efficient indexing
- `utilities::algebraic::Algebraic` - Square notation conversion
- `utilities::board::PrintAsBoard` - Debug visualization

2.5 · Test Plan: Raw Crate

Unit and integration testing for attack table generation, including bishops, rooks, kings, knights, pawns, and path calculations (between/through).

2.5.1 · Test Cases

TC-001: Bishop Attacks

Requirements: Attack generation with blockers

Test	Input	Expected
Empty board	sq=e4, blockers=0	Full diagonal rays
Single blocker	sq=e4, blockers=g6	Stops at g6
Multiple blockers	sq=e4, blockers=d3,f5	Stops at both
Edge squares	sq=a1, blockers=0	Single diagonal
Mask generation	All 64 squares	Excludes edges

TC-002: Rook Attacks

Requirements: Attack generation with blockers

Test	Input	Expected
Empty board	sq=e4, blockers=0	Full rank + file
Rank blocker	sq=e4, blockers=g4	Stops at g4
File blocker	sq=e4, blockers=e7	Stops at e7
Corner square	sq=a1, blockers=0	Rank 1 + file a
Mask generation	All 64 squares	Excludes edges

TC-003: King Attacks

Requirements: Fixed attack patterns

Test	Input	Expected
Center square	sq=e4	8 surrounding squares
Corner square	sq=a1	3 squares (b1,a2,b2)
Edge square	sq=e1	5 squares
All squares	0-63	Valid patterns only

TC-004: Knight Attacks

Requirements: Fixed L-shape patterns

Test	Input	Expected
Center square	sq=e4	8 knight moves
Corner square	sq=a1	2 moves (b3,c2)
Edge square	sq=e1	4 moves
All squares	0-63	Valid L-shapes only

TC-005: Pawn Attacks

Requirements: Color-specific diagonal attacks

Test	Input	Expected
White center	sq=e4, color=White	d5,f5
Black center	sq=e5, color=Black	d4,f4
White edge	sq=a4, color=White	b5 only
Promotion rank	White on rank 7	2 diagonals

TC-006: Between Calculation

Requirements: Exclusive range between squares

Test	Input	Expected
Same rank	e1 to e5	e2,e3,e4
Diagonal	a1 to h8	b2,c3,…,g7
Adjacent squares	e1 to e2	Empty (0)
Not aligned	e1 to c3	Empty (0)

TC-007: Line Through Calculation

Requirements: Full line through two squares

Test	Input	Expected
Same rank	e1 to e5	Full rank 1
Diagonal	e1 to c3	Full a8-h1 diagonal
Adjacent file	e1 to e2	Full e-file
Not aligned	e1 to c4	Empty (0)

TC-008: Attack Table Generation

Requirements: PEXT indexing and lazy initialization

Test	Input	Expected
Table size	ROOK_ATTACKS	64 vectors
Index validity	All blocker combos	Valid attacks
LazyLock init	First access	Tables generated
PEXT lookup	mask + blockers	Correct attack set

TC-009: Memory and Performance

Requirements: Size verification and warmup

Test	Input	Expected
Rook table size	All entries	`~0.5` MB
Bishop table size	All entries	`~0.5` MB
Static tables	King/Knight/Pawn	`<2 KB` each
Warmup time	All tables	`<50ms` debug

2.5.2 · Environment

- Rust toolchain: stable with x86_64 target
- Required features: BMI2 (PEXT instruction)
- Test framework: built-in `cargo test`

3 · Utilities Crate

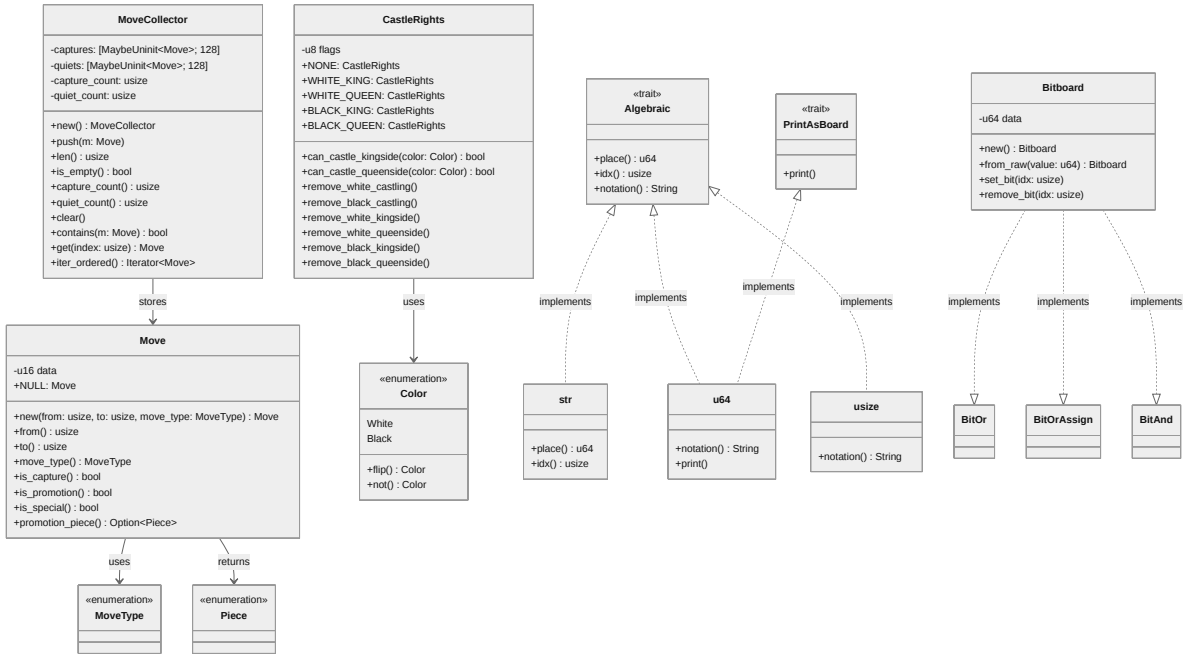


Figure 5: Class Diagram For The `utilities` crate

3.1 · Introduction

3.1.1 · Purpose

This section specifies the software requirements for the `utilities` crate of the chess engine. The `utilities` crate provides helper functions and traits for working with algebraic notation and board visualization.

3.1.2 · Scope

The `utilities` crate serves as a utility layer, providing convenient conversion functions between algebraic chess notation and bitboard representations, as well as debugging visualization tools.

3.2 · Functional Requirements

ID	Description	Type	Status
UTIL-F-001	Algebraic trait must parse comma-separated square notation strings (e.g., “g2,g4,g5”) and return u64 bitboard with corresponding bits set	Functional	Complete

UTIL-F-002	`place()` method must validate and skip invalid square notations (wrong file/rank format, out of bounds)	Functional	Complete
UTIL-F-003	`place()` method must handle whitespace in input strings and empty comma-separated values gracefully	Functional	Complete
UTIL-F-004	`idx()` method must convert single algebraic notation square (e.g., “g2”) to 0-63 board index	Functional	Complete
UTIL-F-005	`idx()` method must support case-insensitive file letters (a-h or A-H)	Functional	Complete
UTIL-F-006	`notation()` method must convert u64 bitboard to comma-separated algebraic notation string of set bits	Functional	Complete
UTIL-F-007	`notation()` method must iterate through bitboard using `trailing_zeros()` optimization for sparse boards	Functional	Complete
UTIL-F-008	`notation()` method must be implemented for both u64 and usize types	Functional	Complete
UTIL-F-009	`PrintAsBoard` trait must render u64 bitboard as visual 8x8 ASCII chessboard	Functional	Complete
UTIL-F-010	`print()` method must display set bits as ‘X’ and unset bits as ‘.’ with rank/file labels	Functional	Complete
UTIL-F-011	`print()` method must display board with ranks 8-1 (top to bottom) and files a-h (left to right)	Functional	Complete
UTIL-F-012	`print()` method must only compile in debug builds `(cfg(debug_assertions))` to avoid bloat in release	Functional	Complete
UTIL-F-013	Algebraic trait must be implemented for str type with `place()` and `idx()` methods	Functional	Complete
UTIL-F-014	Algebraic trait must be implemented for u64 and usize types with `notation()` method	Functional	Complete
UTIL-F-015	`place()` must correctly map algebraic squares to bitboard indices: a1=0, h1=7, a8=56, h8=63	Functional	Complete
UTIL-F-016	System must provide test suite validating `place()`, `idx()`, and `notation()` conversions	Functional	Complete

3.3 · Non-Functional Requirements

ID	Description	Type	Status
UTIL-NF-001	`place()` parsing must be resilient to malformed input without panicking	Non-Functional	Complete
UTIL-NF-002	`print()` method must be zero-cost in release builds through conditional compilation	Non-Functional	Complete

3.4 · Dependencies

- `std::string::String` - String building for notation() output

- Internal module structure: `algebraic` and `board` submodules

4 · Zobrist Crate

4.1 · Sequence Diagram

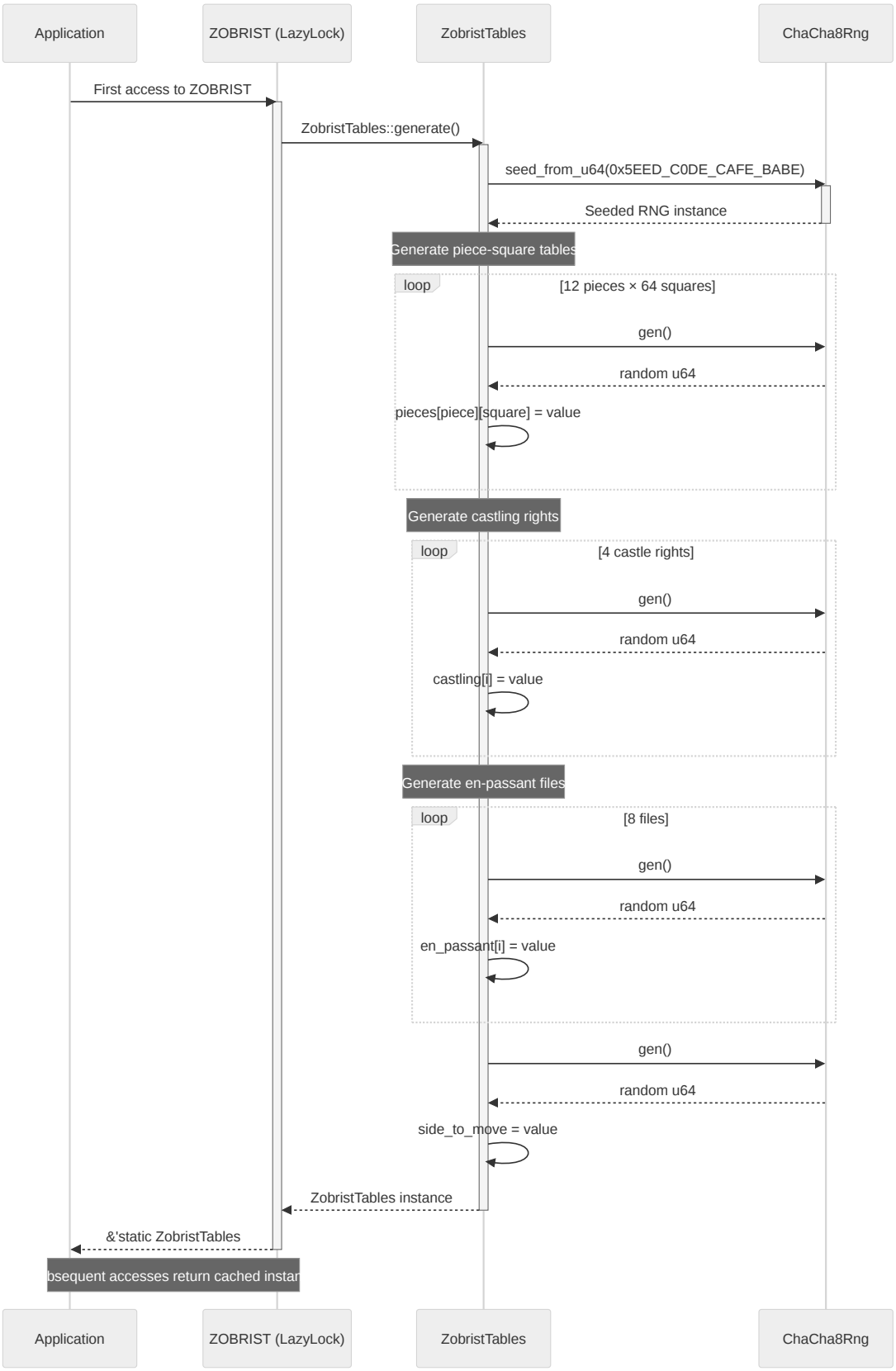


Figure 6: Sequence Diagram For The `Zobrist` crate

4.2 · Class Relationship Diagram

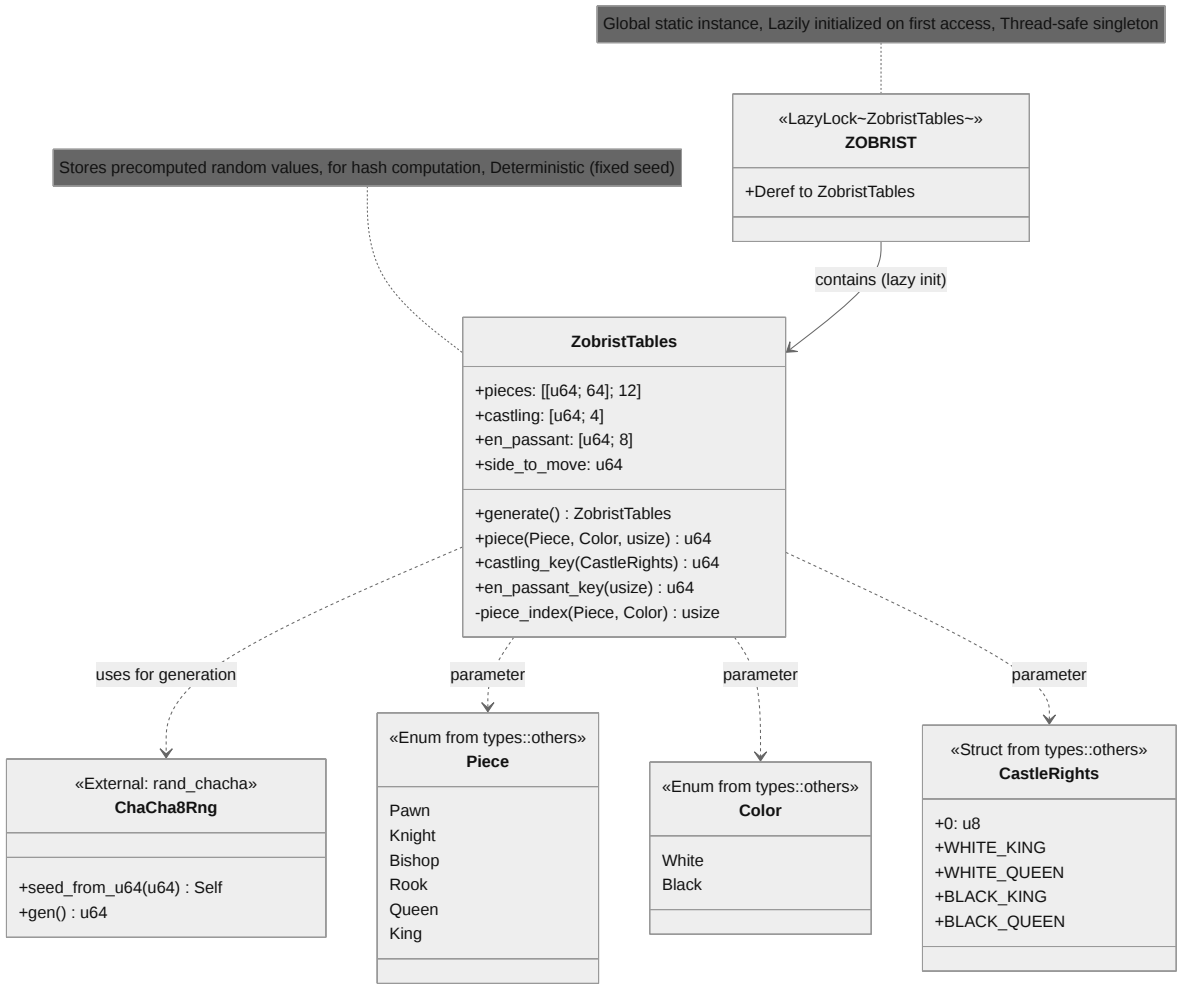


Figure 7: Class Diagram For The `zobrist` crate

4.3 · Introduction

4.3.1 · Purpose

This section specifies the software requirements for the `zobrist` crate of the chess engine. The `zobrist` crate provides Zobrist hashing functionality for efficient position identification and transposition table lookups.

4.3.2 · Scope

The `zobrist` crate implements Zobrist hashing, a technique that assigns random numbers to board components and uses XOR operations to create unique hash keys for chess positions, enabling fast position comparison and transposition detection.

4.4 · Functional Requirements

ID	Description	Type	Status
ZOBRIST-F-001	ZobristTables must generate random u64 values for all 12 piece types (6 pieces × 2 colors) across 64 squares	Functional	Complete
ZOBRIST-F-002	System must use deterministic random number generation with fixed seed (0x5EED_C0DE_CAFE_BABE; just a fun seed that can be represented in hex) for reproducibility	Functional	Complete
ZOBRIST-F-003	ZobristTables must generate 4 unique random values for castling rights (White King/Queen, Black King/Queen)	Functional	Complete
ZOBRIST-F-004	ZobristTables must generate 8 unique random values for en passant file possibilities (a-h)	Functional	Complete
ZOBRIST-F-005	ZobristTables must generate single random value for side-to-move indicator	Functional	Complete
ZOBRIST-F-006	piece() method must return Zobrist key for given piece type, color, and square index	Functional	Complete
ZOBRIST-F-007	piece_index() must map piece and color to table index: White pieces (0-5), Black pieces (6-11)	Functional	Complete
ZOBRIST-F-008	castling_key() must compute XOR combination of relevant castling rights flags	Functional	Complete
ZOBRIST-F-009	castling_key() must check each of 4 castling rights independently and XOR corresponding values	Functional	Complete
ZOBRIST-F-010	en_passant_key() must return Zobrist value for specified en passant file (0-7)	Functional	Complete
ZOBRIST-F-011	System must provide global static ZOBRIST instance via LazyLock for singleton access	Functional	Complete
ZOBRIST-F-012	ZOBRIST singleton must initialize exactly once on first access using lazy evaluation	Functional	Complete

4.5 · Non-Functional Requirements

ID	Description	Type	Status
ZOBRIST-NF-001	All lookup methods (piece, castling_key, en_passant_key) must be inlined for zero-overhead access	Non-Functional	Complete
ZOBRIST-NF-002	Random number generation must use ChaCha8Rng for cryptographic quality and reproducibility	Non-Functional	Complete

ZOBRIST-NF-003	Zobrist tables must remain constant after initialization for thread-safe concurrent access	Non-Functional	Complete
ZOBRIST-NF-004	Memory footprint must be minimal: 12×64×8 bytes (pieces) + 4×8 (castling) + 8×8 (en passant) + 8 (side) = 6,216 bytes	Non-Functional	Complete
ZOBRIST-NF-005	System must use LazyLock to defer initialization cost until first use	Non-Functional	Complete
ZOBRIST-NF-006	castling_key() must use bitwise operations for efficient flag checking without branches where possible	Non-Functional	Complete

4.6 · Dependencies

- `rand` - Random number generation trait
- `rand_chacha::ChaCha8Rng` - ChaCha8 PRNG implementation
- `std::sync::LazyLock` - Thread-safe lazy initialization
- `types::others::{CastleRights, Color, Piece}` - Core type definitions

5 · Board Crate

5.0.1 · Class Relationship

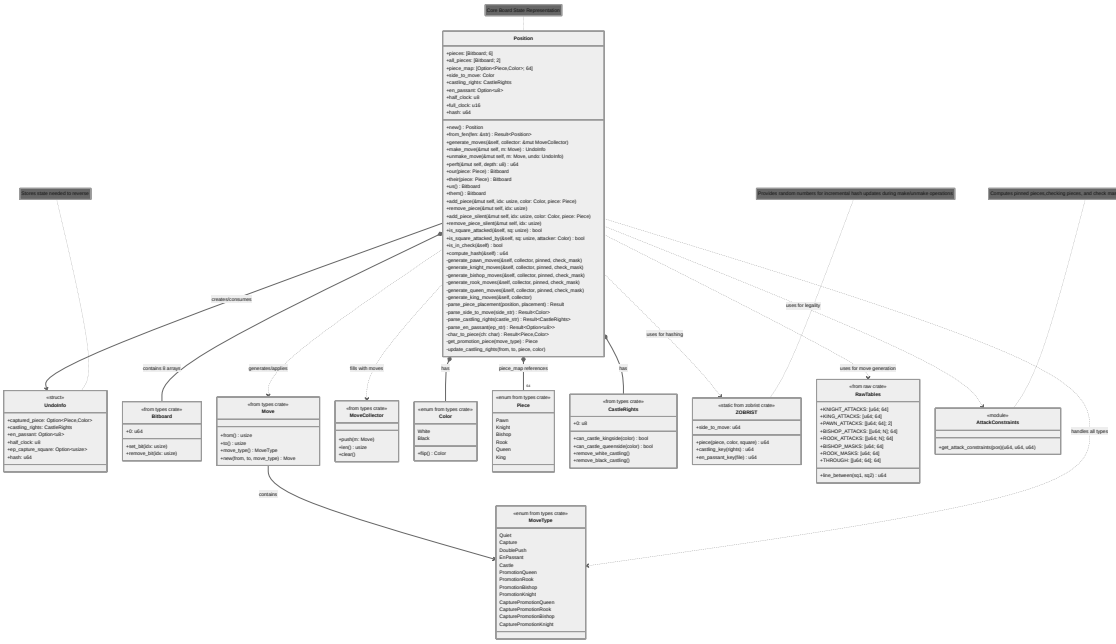
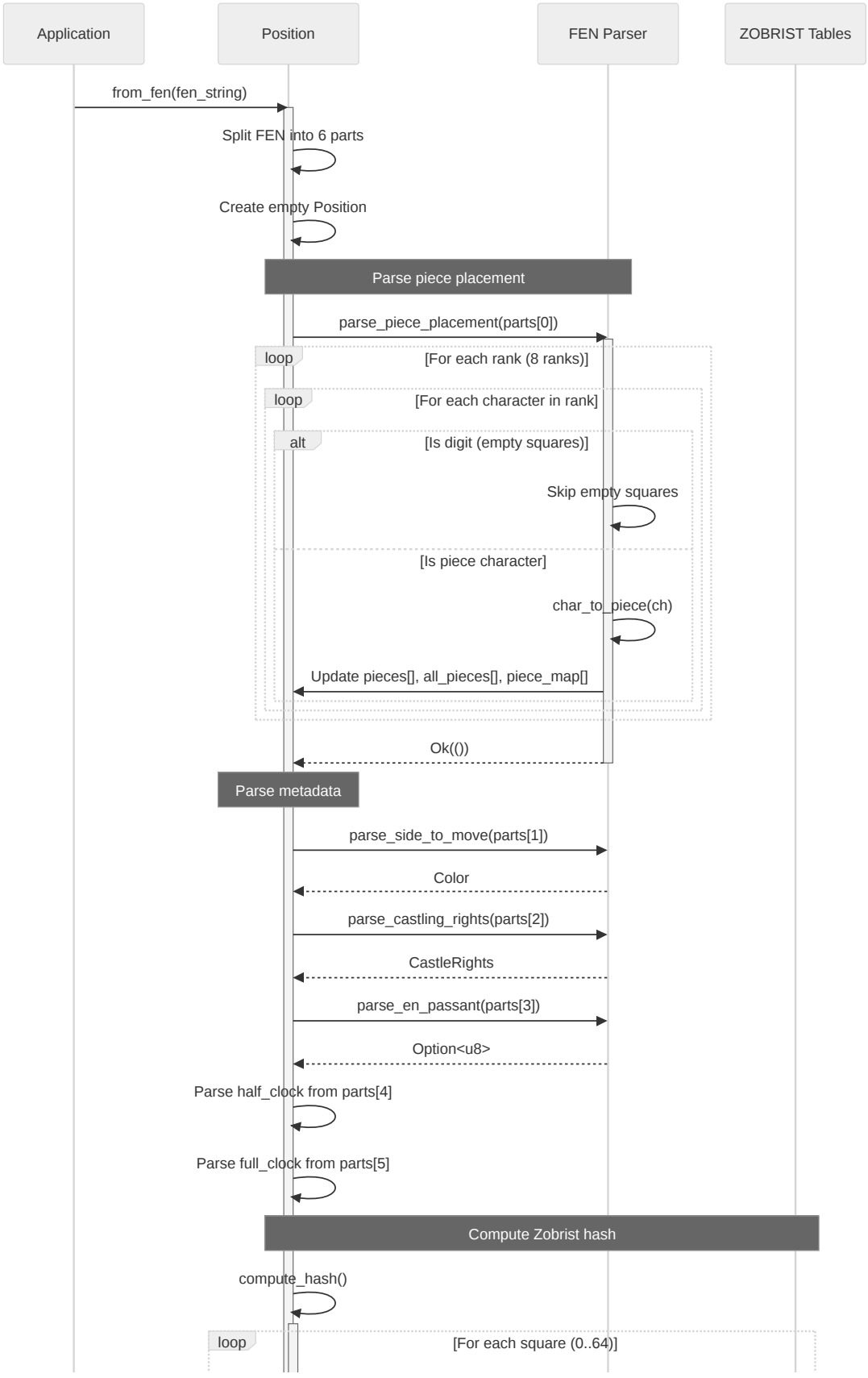


Figure: Sequence Diagram For The Board Crate

5.0.2 · Position Initialization From FEN Sequence



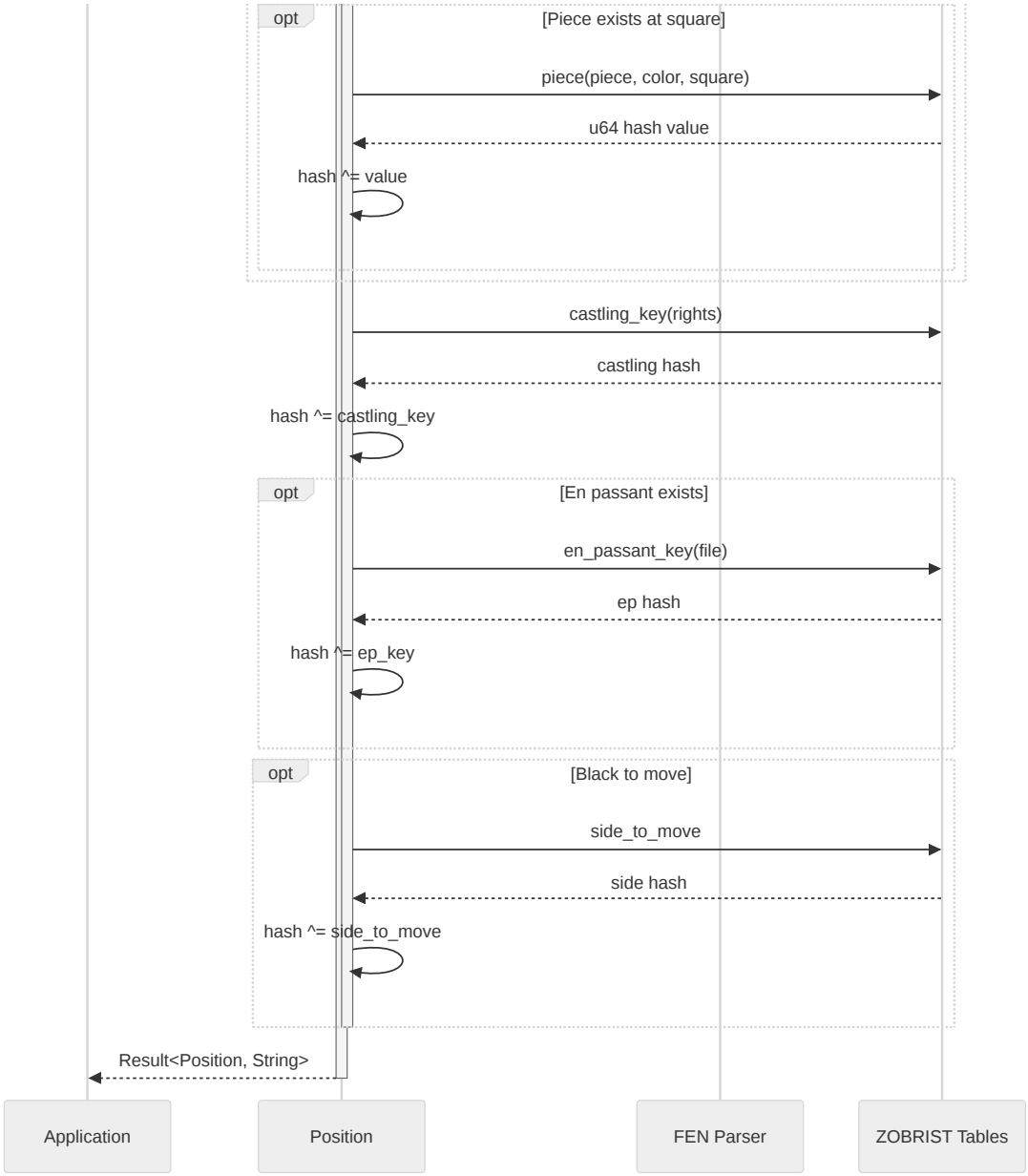
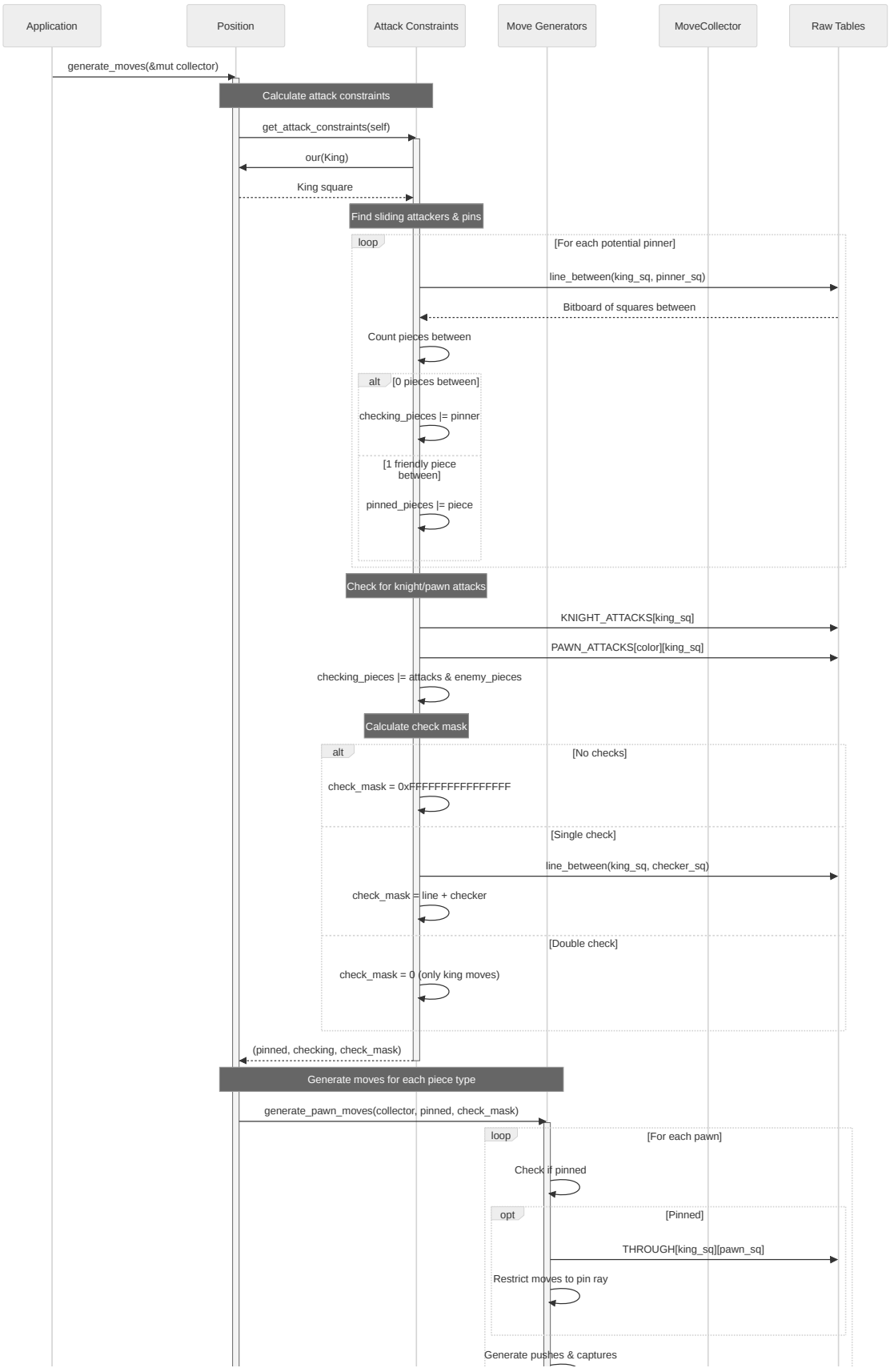
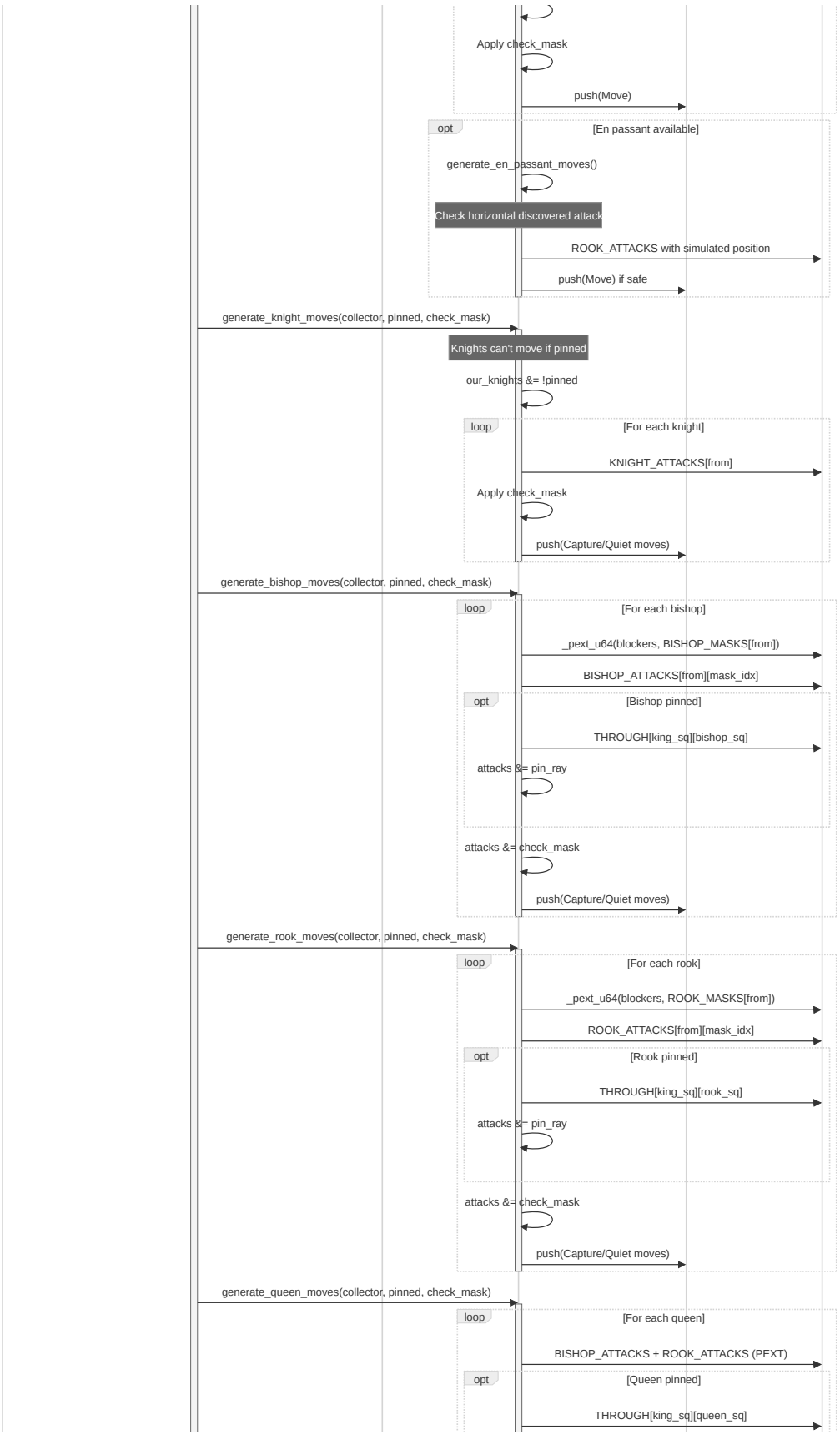


Figure: Sequence Diagram For The Board Crate, Parsing FEN

5.0.3 · Move Generation Sequence




Apply check_mask
push(Move)
opt
[En passant available]
generate_en_passant_moves()
Check horizontal discovered attack
ROOK_ATTACKS with simulated position
push(Move) if safe
generate_knight_moves(collector, pinned, check_mask)
Knights can't move if pinned
our_knights &= !pinned
loop
[For each knight]
KNIGHT_ATTACKS[from]
Apply check_mask
push(Capture/Quiet moves)
generate_bishop_moves(collector, pinned, check_mask)
loop
[For each bishop]
_pext_u64(blockers, BISHOP_MASKS[from])
BISHOP_ATTACKS[from][mask_idx]
opt
[Bishop pinned]
THROUGH[king_sq][bishop_sq]
attacks &= pin_ray
attacks &= check_mask
push(Capture/Quiet moves)
generate_rook_moves(collector, pinned, check_mask)
loop
[For each rook]
_pext_u64(blockers, ROOK_MASKS[from])
ROOK_ATTACKS[from][mask_idx]
opt
[Rook pinned]
THROUGH[king_sq][rook_sq]
attacks &= pin_ray
attacks &= check_mask
push(Capture/Quiet moves)
generate_queen_moves(collector, pinned, check_mask)
loop
[For each queen]
BISHOP_ATTACKS + ROOK_ATTACKS (PEXT)
opt
[Queen pinned]
THROUGH[king_sq][queen_sq]

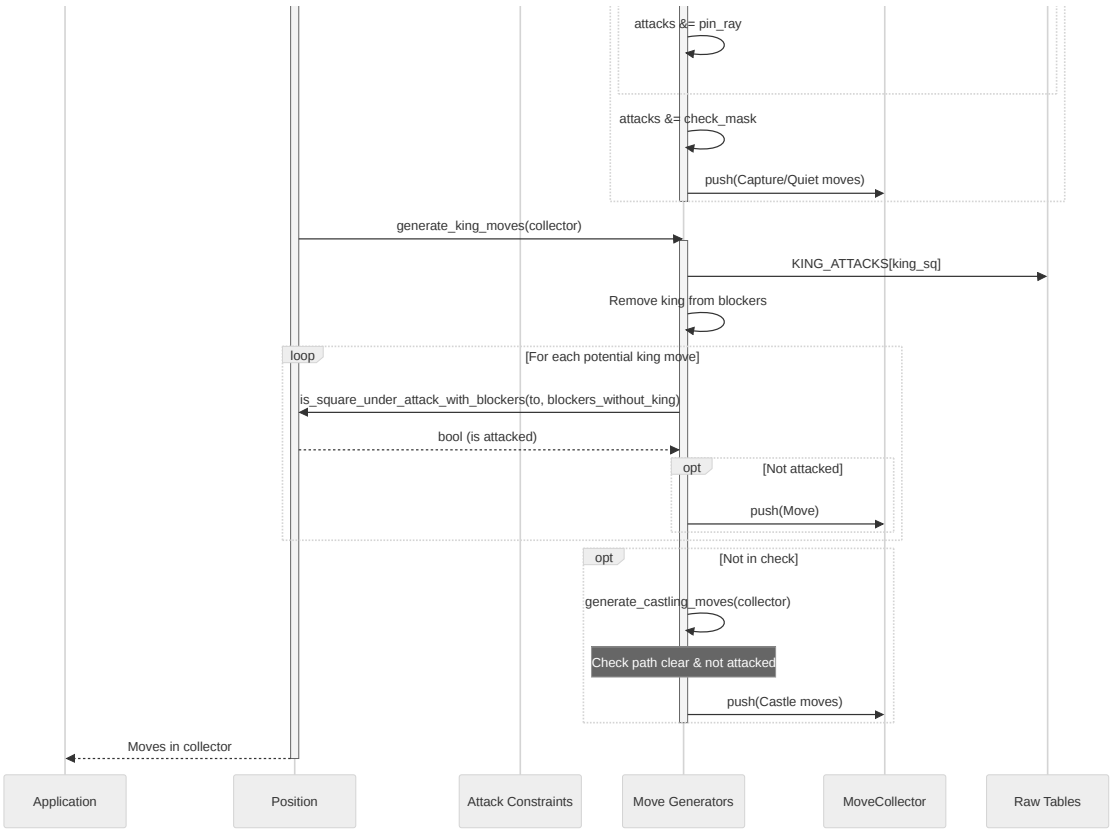
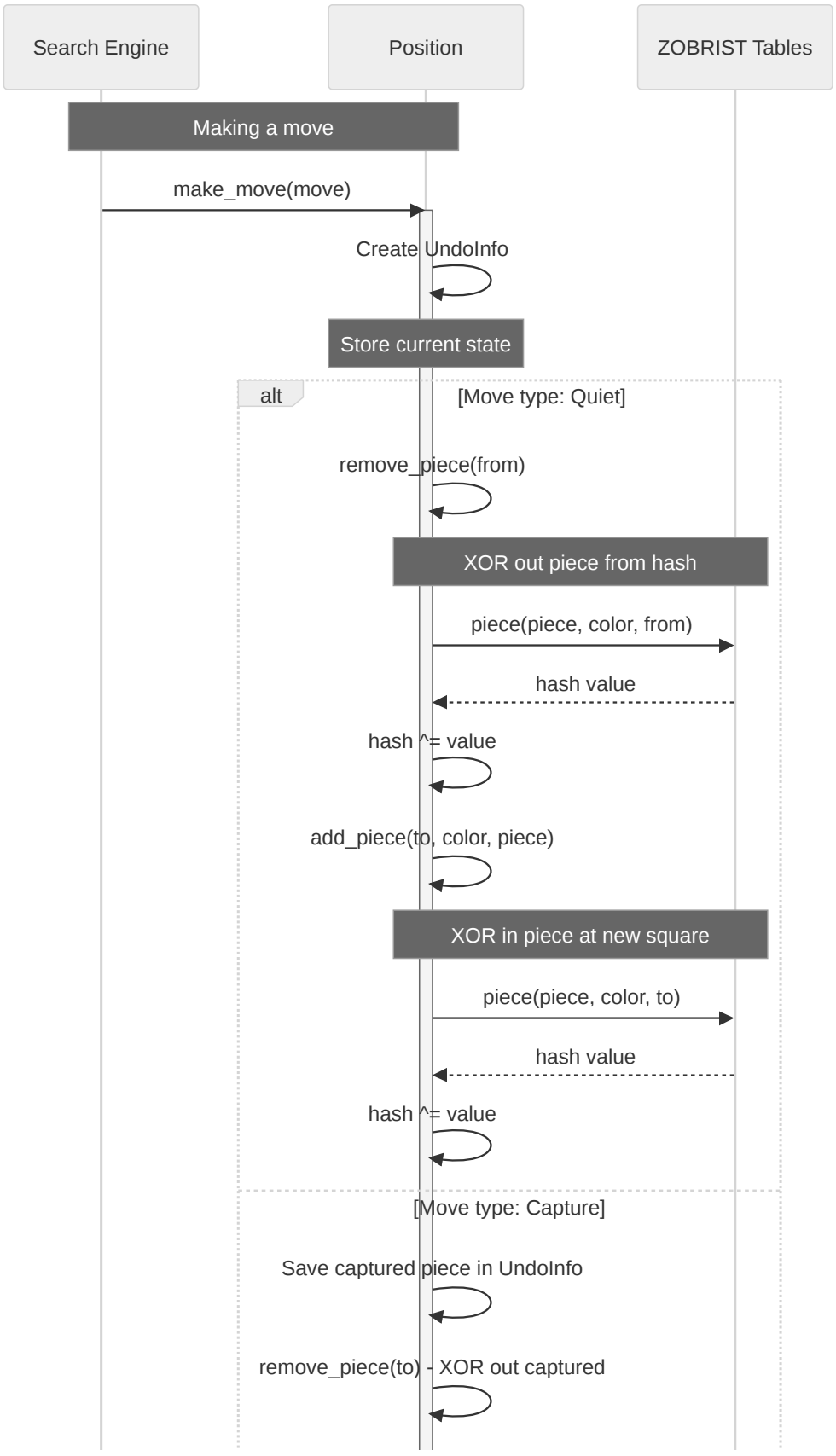
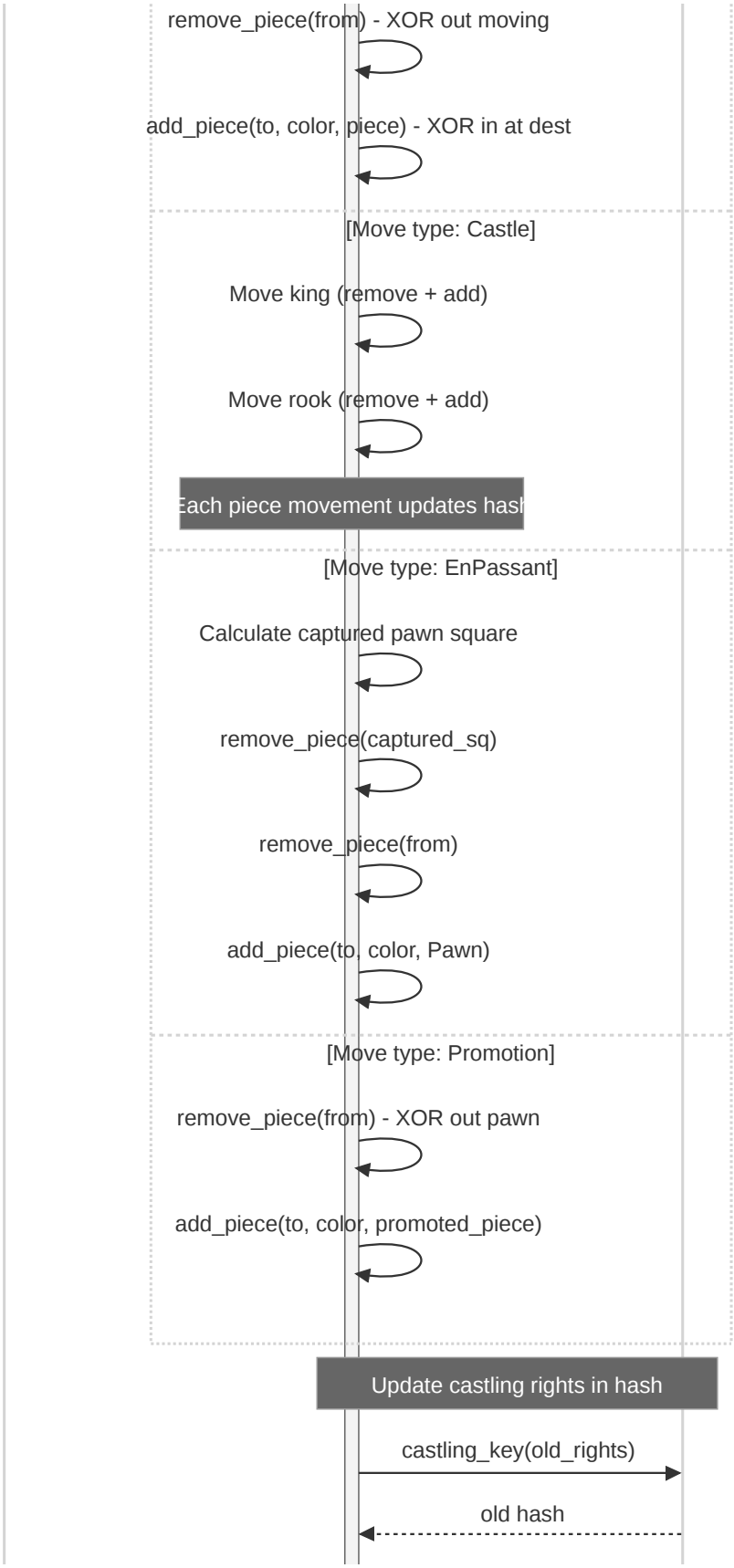
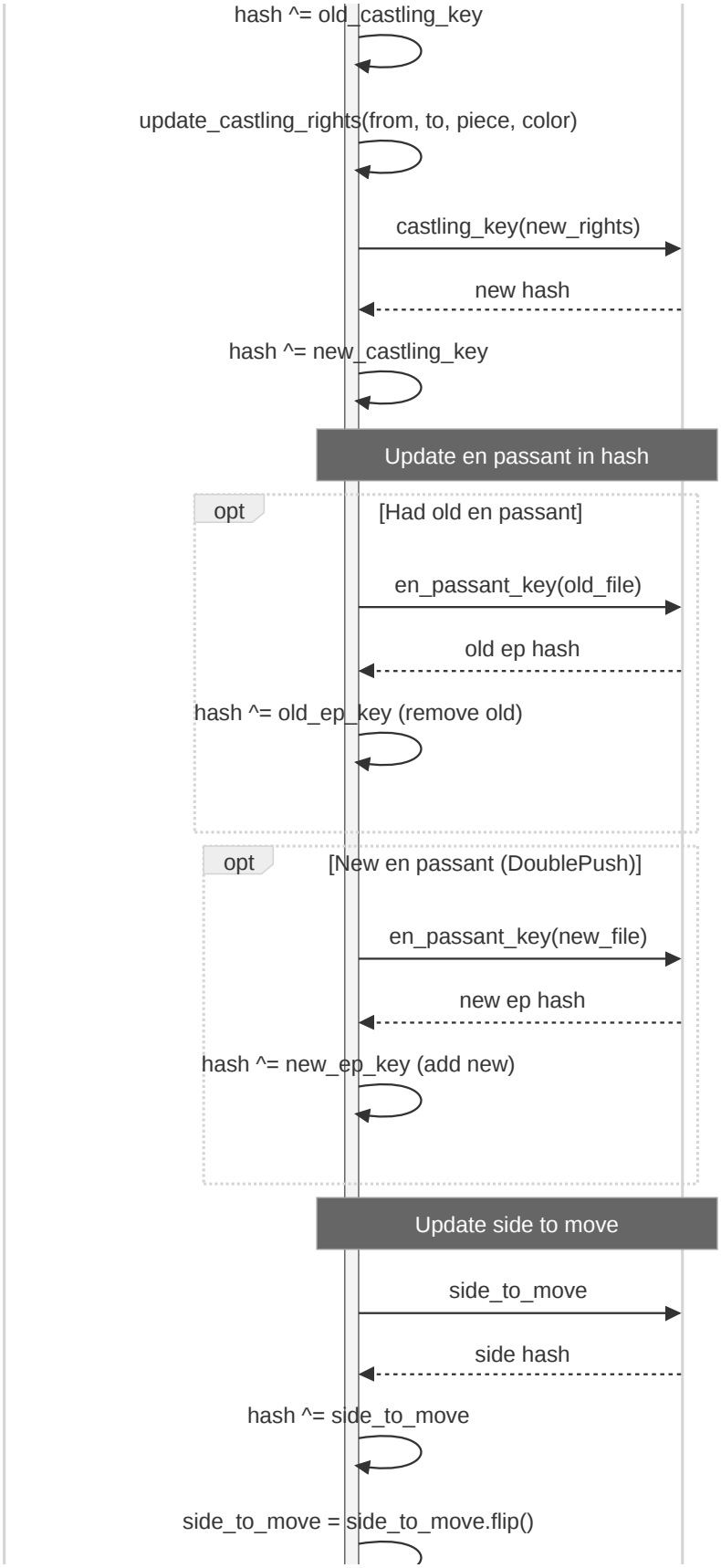


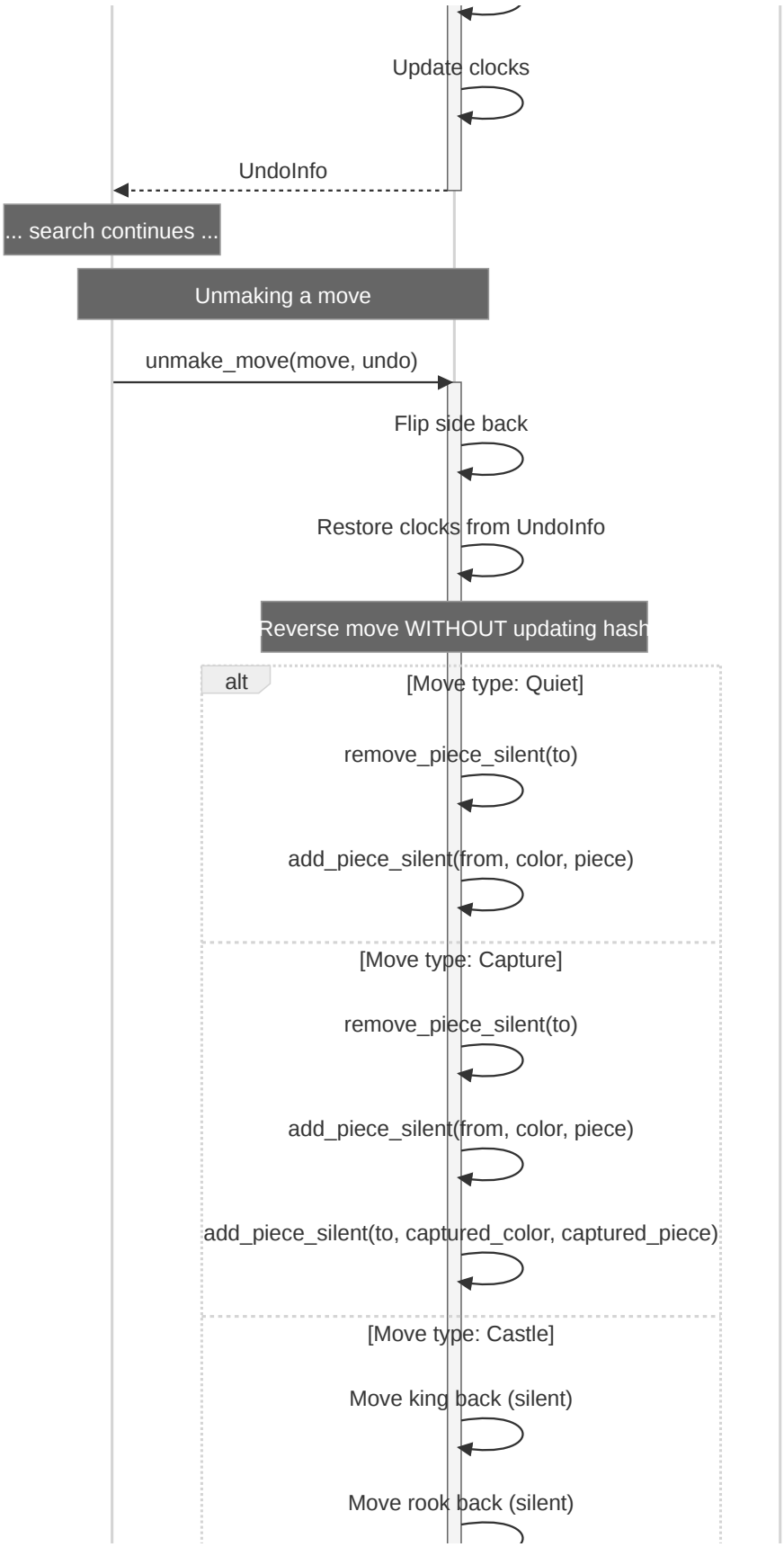
Figure: Sequence Diagram For The Board Crate, Make/Unmake

5.0.4 · Make/ Unmake Sequence




remove_piece(from) - XOR out moving
add_piece(to, color, piece) - XOR in at dest
[Move type: Castle]
Move king (remove + add)
Move rook (remove + add)
Each piece movement updates hash
[Move type: EnPassant]
Calculate captured pawn square
remove_piece(captured_sq)
remove_piece(from)
add_piece(to, color, Pawn)
[Move type: Promotion]
remove_piece(from) - XOR out pawn
add_piece(to, color, promoted_piece)
Update castling rights in hash
castling_key(old_rights)
old hash


hash ^= old_castling_key
update_castling_rights(from, to, piece, color)
castling_key(new_rights)
new hash
hash ^= new_castling_key
Update en passant in hash
opt
[Had old en passant]
en_passant_key(old_file)
old ep hash
hash ^= old_ep_key (remove old)
opt
[New en passant (DoublePush)]
en_passant_key(new_file)
new ep hash
hash ^= new_ep_key (add new)
Update side to move
side_to_move
side hash
hash ^= side_to_move
side_to_move = side_to_move.flip()


Update clocks
UndoInfo
... search continues ...
Unmaking a move
unmake_move(move, undo)
Flip side back
Restore clocks from UndoInfo
Reverse move WITHOUT updating hash
alt
[Move type: Quiet]
remove_piece_silent(to)
add_piece_silent(from, color, piece)
[Move type: Capture]
remove_piece_silent(to)
add_piece_silent(from, color, piece)
add_piece_silent(to, captured_color, captured_piece)
[Move type: Castle]
Move king back (silent)
Move rook back (silent)

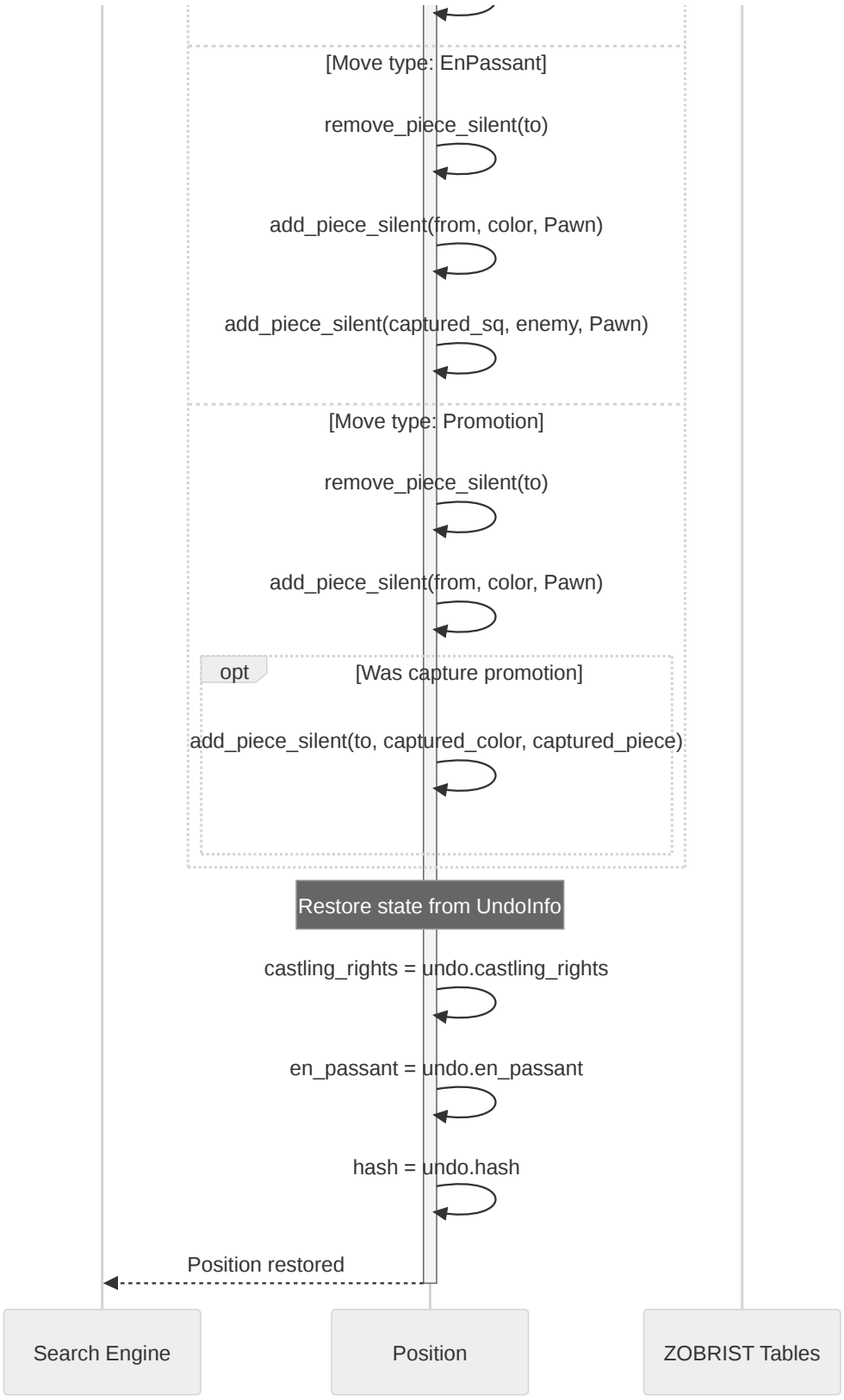


Figure: Sequence Diagram For The Board Crate, Move Generation

5.1 · Introduction

5.1.1 · Purpose

This section specifies the software requirements for the `board` crate of the chess engine. The `board` crate provides the core Position representation and move generation functionality for legal chess moves.

5.1.2 · Scope

The `board` crate implements a bitboard-based chess position representation with efficient move generation for all piece types, FEN parsing, Zobrist hashing integration, and legality checking including pins, checks, and special moves.

5.2 · Functional Requirements

ID	Description	Type	Status
BOARD-F-001	Position must maintain 6 bitboards for piece types (Pawn, Knight, Bishop, Rook, Queen, King) and 2 bitboards for colors	Functional	Complete
BOARD-F-002	Position must maintain piece_map array for O(1) piece identification at any square (0-63)	Functional	Complete
BOARD-F-003	from_fen() must parse standard FEN strings with 6 parts: piece placement, side to move, castling rights, en passant, halfmove clock, fullmove number	Functional	Complete
BOARD-F-004	FEN parser must validate rank count (8 ranks), file count (8 files per rank), and piece characters (PNBRQKpnbrqk)	Functional	Complete
BOARD-F-005	Move generation must handle pin masks to restrict pinned piece movement along pin rays using THROUGH lookup table	Functional	Complete
BOARD-F-006	Move generation must apply check masks to ensure moves block or capture checking pieces	Functional	Complete
BOARD-F-007	Pawn move generation must handle single push, double push, captures, promotions (4 types), and en passant with horizontal discovered check detection	Functional	Complete
BOARD-F-008	Knight move generation must exclude pinned knights entirely (knights cannot move when pinned)	Functional	Complete
BOARD-F-009	Sliding piece move generation (Bishop, Rook, Queen) must use PEXT-based magic bitboard lookups with BISHOP_ATTACKS and ROOK_ATTACKS tables	Functional	Complete

BOARD-F-010	King move generation must verify target squares are not under attack using is_square_under_attack_with_blockers() with king removed from board	Functional	Complete
BOARD-F-011	Castling move generation must verify: castling rights, empty squares between king and rook, king not in check, intermediate squares not attacked	Functional	Complete
BOARD-F-012	System must provide add_piece() and remove_piece() methods that update bitboards, piece_map, and Zobrist hash atomically	Functional	Complete
BOARD-F-013	System must provide silent variants (add_piece_silent, remove_piece_silent) for unmake_move that skip hash updates	Functional	Complete
BOARD-F-014	compute_hash() must calculate full Zobrist hash from current position: pieces, castling rights, en passant file, side to move	Functional	Complete
BOARD-F-015	is_in_check() must detect if current side's king is under attack, is_enemy_in_check() must check opponent's king	Functional	Complete

5.3 · Non-Functional Requirements

ID	Description	Type	Status
BOARD-NF-001	All move generation and board query methods must be inlined for zero-overhead access	Non-Functional	Complete
BOARD-NF-002	Sliding piece attack generation must use PEXT instruction (`_pext_u64`) for optimal performance on modern x86_64 CPUs	Non-Functional	Complete
BOARD-NF-003	Move generation must use bitboard iteration (trailing_zeros, LSB pop) for efficient square enumeration	Non-Functional	Complete
BOARD-NF-004	Position struct must be Clone and PartialEq for transposition table usage and position comparison	Non-Functional	Complete
BOARD-NF-005	Memory footprint must be minimal: 6×8 (pieces) + 2×8 (colors) + 64×3 (piece_map) + 18 (metadata) ≈ 274 bytes per position	Non-Functional	Complete

5.4 · Dependencies

- `zobrist` - Zobrist hashing tables

- `raw` - Precomputed attack tables (PAWN_ATTACKS, KNIGHT_ATTACKS, KING_ATTACKS, BISHOP_ATTACKS/MASKS, ROOK_ATTACKS/MASKS, THROUGH)
- `types::bitboard::Bitboard` - Bitboard wrapper type
- `types::others::{CastleRights, Color, Piece}` - Core type definitions
- `types::moves::{Move, MoveCollector, MoveType}` - Move representation
- `std::arch::x86_64::_pext_u64` - PEXT instruction for magic bitboards

5.5 · Test Plan: Board Crate

Unit and integration testing for the Position struct including FEN parsing, move generation, make/unmake, perft validation, and legality checking.

5.5.1 · Test Cases

TC-001: FEN Parsing

Requirements: Position initialization from FEN

Test	Input FEN	Expected
Starting position	Standard FEN	All pieces correct
Missing parts	4-part FEN	Error
Invalid pieces	FEN with ‘X’	Error
Castling rights	`KQkq`, `Kq`, `-`	Parsed correctly
En passant	`e3`, `-`	Parsed correctly

TC-002: Move Generation - Pawns

Requirements: Legal pawn moves including promotions and en passant

Test	Position	Expected Moves
Starting position	Initial FEN	16 moves
En passant	Valid EP square	EP move included
Promotion	Pawn on 7th rank	4 promotion types
Capture promotion	Capture on 8th	4 capture promos
Diagonal pin	Pawn pinned diagonally	0 moves

TC-003: Move Generation - Knights

Requirements: Legal knight moves respecting pins

Test	Position	Expected
Starting position	Initial FEN	4 moves (b1/g1)
Center knight	Knight on e4	Up to 8 moves
Pinned knight	Knight pinned to king	0 moves
Check evasion	In check	Blocking moves only

TC-004: Move Generation - Bishops

Requirements: Legal bishop moves with blocker interaction

Test	Position	Expected
Starting position	Initial FEN	0 moves
Open diagonals	Bishops on d5,e3	17 moves
Both pinned	Bishops pinned	0 moves
Pin ray movement	Bishop on pin ray	Moves along ray only

TC-005: Move Generation - Rooks

Requirements: Legal rook moves with PEXT lookup

Test	Position	Expected
Starting position	Initial FEN	0 moves
Open files	Rook on a1 clear	15 moves
Pinned rooks	Multiple pins	Moves along pin ray
Capture + quiet	Mixed targets	Separated correctly

TC-006: Move Generation - Queens

Requirements: Combined rook + bishop attacks

Test	Position	Expected
Starting position	Initial FEN	0 moves
Open board	Queen on d1	10 moves
Pinned queen	Queen pinned	Ray moves only
Multiple queens	Several queens	All moves correct

TC-007: Move Generation - Kings

Requirements: King moves and castling

Test	Position	Expected
Starting position	Initial FEN	0 moves
Kingside castle	Rights + clear	Castle move
Attacked squares	f1 under attack	No castle
In check	King in check	No castle allowed
Blockers removed	Clear path	King moves valid

TC-008: Make/Unmake Moves

Requirements: Reversible move execution

Test	Scenario	Expected
Quiet move	Make then unmake	Position restored
Capture	Make then unmake	Captured piece restored

Castle	Make then unmake	Rook + king restored
En passant	Make then unmake	Captured pawn restored
Promotion	Make then unmake	Pawn restored
Hash consistency	All move types	Hash matches

TC-009: Attack Detection

Requirements: is_square_attacked correctness

Test	Scenario	Expected
Rook attack	Rook on file	Attacked=true
Blocked attack	Piece blocking	Attacked=false
Knight attack	Knight L-shape	Attacked=true
Pawn attack	Diagonal squares	Attacked=true
Multiple attackers	Several pieces	Attacked=true

TC-010: Pin and Check Detection

Requirements: get_attack_constraints accuracy

Test	Scenario	Expected
No pins/checks	Starting position	Full check_mask
Single pin	One pinned piece	Pin mask set
Single check	One checker	Check mask path
Double check	Two checkers	check_mask=0
Pin + check	Both conditions	Both masks set

TC-011: Perft Validation

Requirements: Move generation correctness via node count

Position	Depth	Expected Nodes
Starting	5	4,865,609
Kiwipete	5	193,690,690
Position 3	6	11,030,083
Position 4	5	15,833,292
Position 6	6	71,179,139

TC-012: Zobrist Hashing

Requirements: Hash consistency and uniqueness

Test	Scenario	Expected
Initial hash	compute_hash()	Non-zero value
Incremental	Make move	Hash updated
Make/unmake	Full cycle	Hash restored

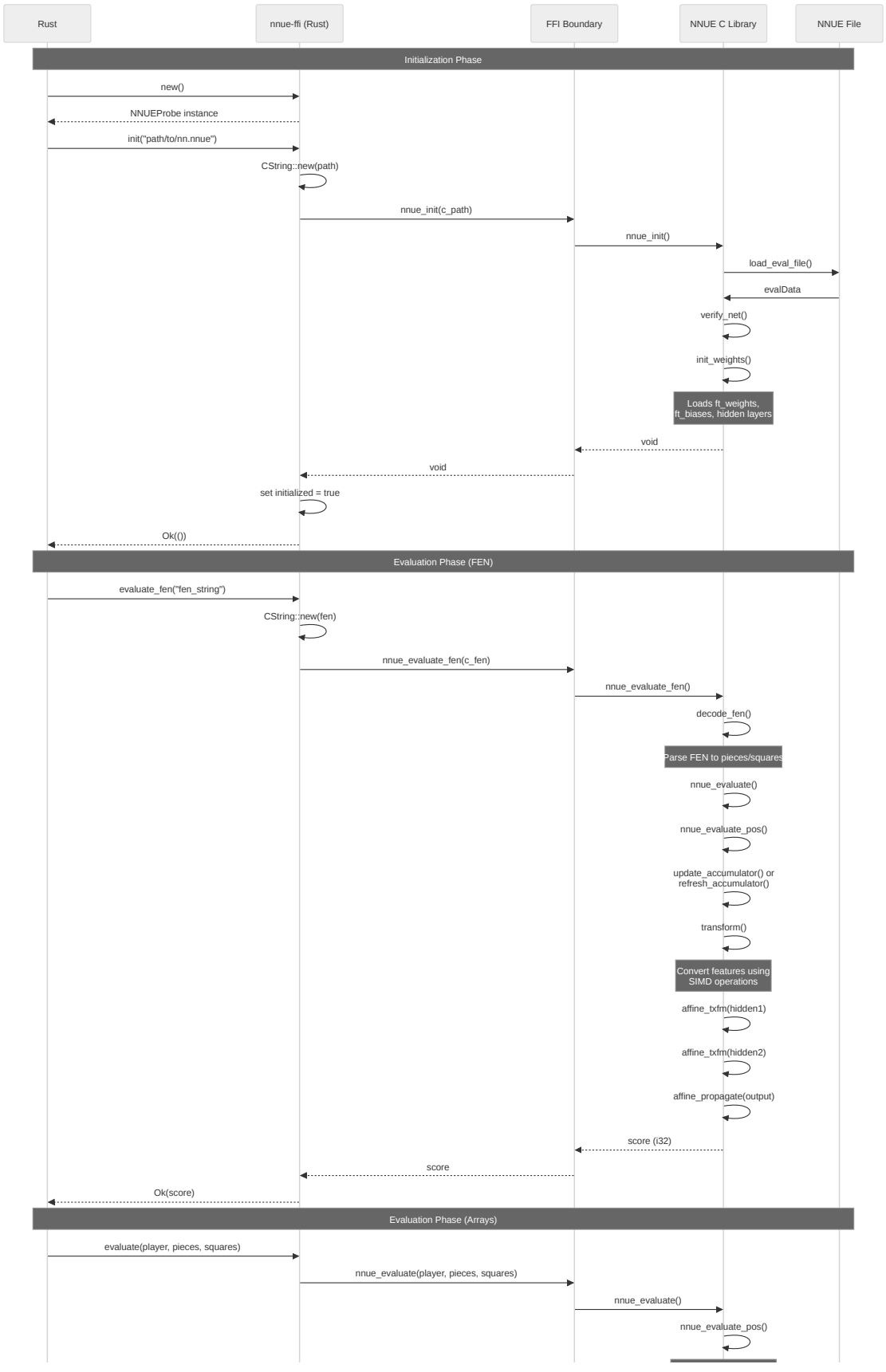
Same position	Different paths	Same hash

5.5.2 · Environment

- Rust toolchain: stable with BMI2 support
- Test framework: built-in `cargo test --lib board`
- Perft tests: `cargo test --release --ignored`

6 · NNUE FFI

6.0.1 · Sequence



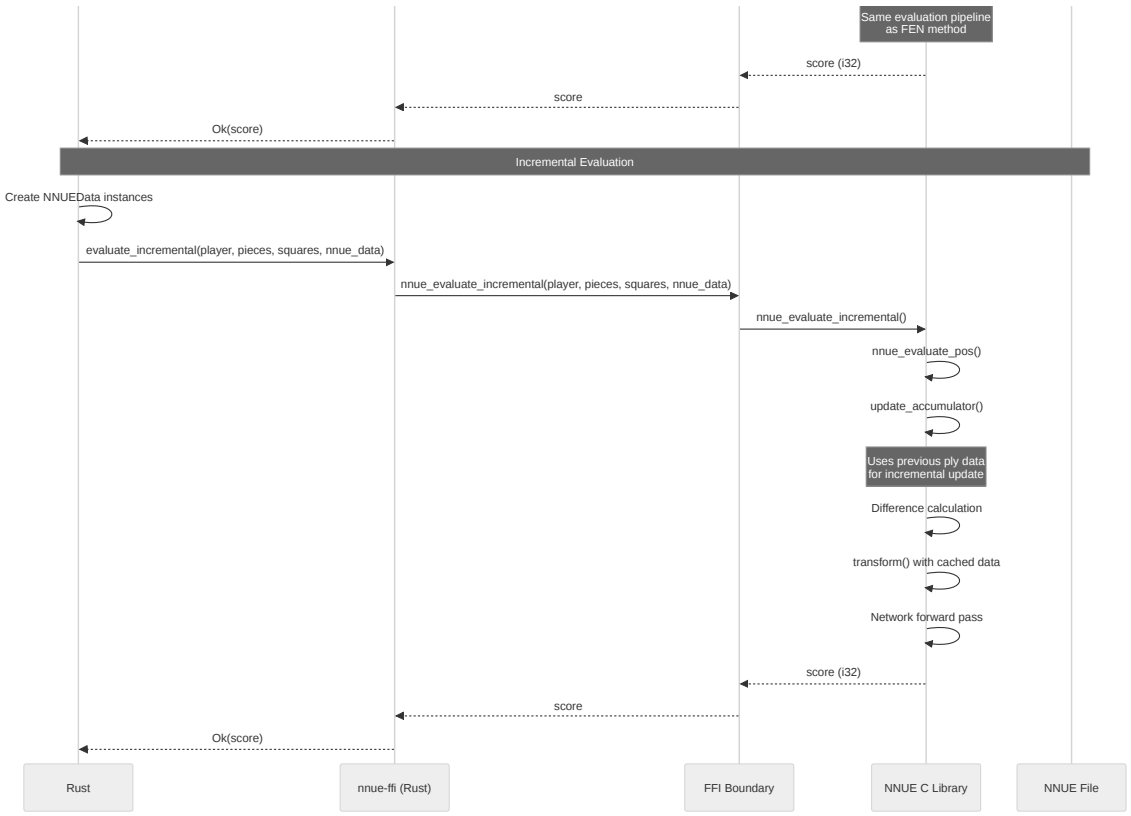


Figure: Sequence Diagram For The nnue-ffi Crate

6.0.2 · Class Relationship

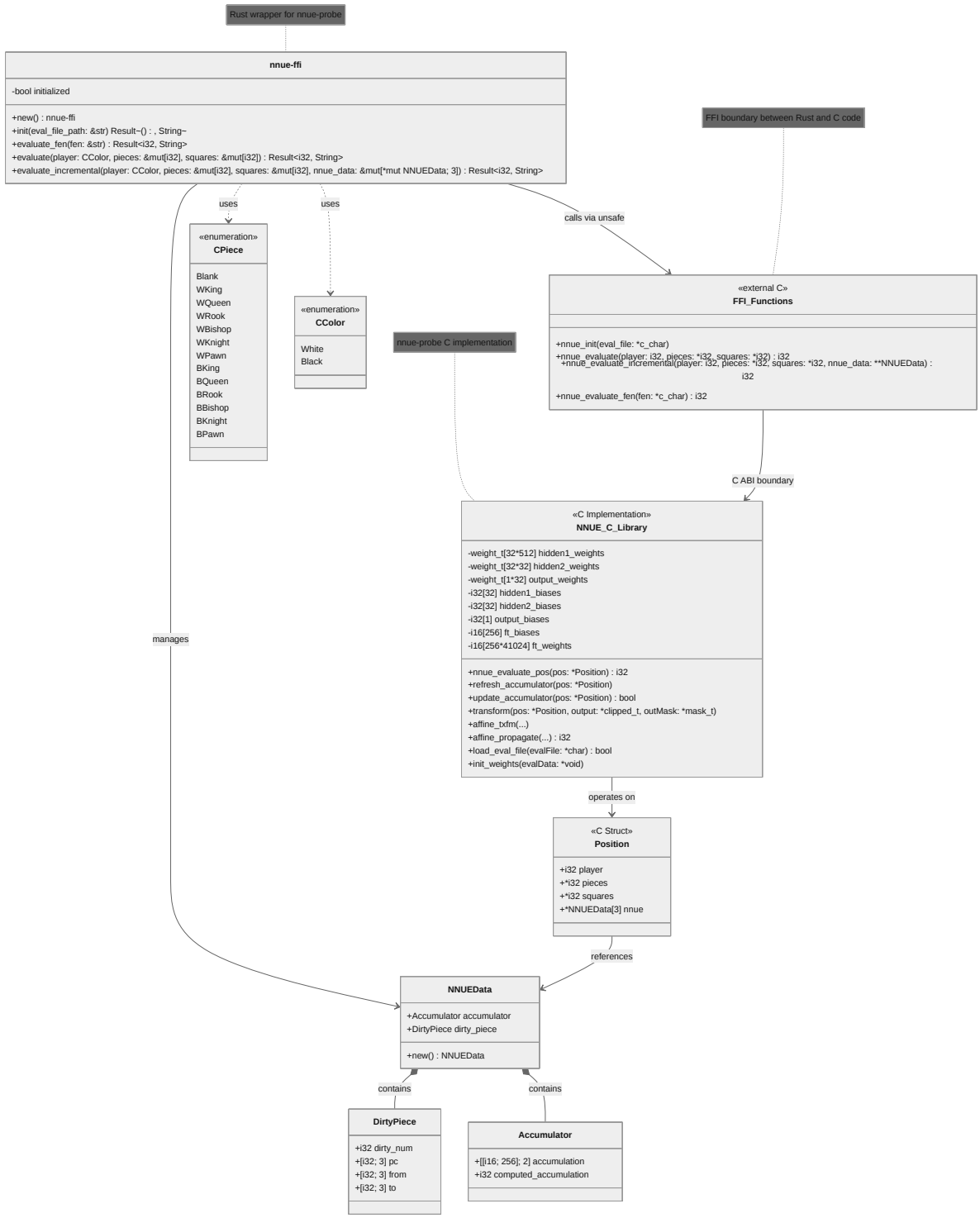


Figure: Class Diagram For The nnue-ffi Crate

6.1 · Introduction

6.1.1 · Purpose

This section specifies the software requirements for the `nnue-probe` crate of the chess engine. The `nnue-probe` crate provides Foreign Function Interface (FFI) bindings to an external NNUE (Efficiently Updatable Neural Network) evaluation library for position scoring.

6.1.2 · Scope

The `nnue-probe` crate implements safe Rust wrappers around a precompiled NNUE library, providing chess position evaluation using neural network inference with support for both standard and incremental evaluation modes.

6.2 · Functional Requirements

ID	Description	Type	Status
NNUE-F-001	NNUEProbe must initialize library exactly once per process using Once synchronization primitive	Functional	Complete
NNUE-F-002	init() must load NNUE evaluation file (.nnue format) from provided path	Functional	Complete
NNUE-F-003	evaluate_fen() must accept FEN string and return evaluation score in centipawns relative to side to move	Functional	Complete
NNUE-F-004	evaluate() must accept piece array, square array (A1=0…H8=63), and CColor, returning centipawns score	Functional	Complete
NNUE-F-005	Piece arrays must follow format: pieces[0]=white king, pieces[1]=black king, pieces[n+1]=0 as terminator	Functional	Complete
NNUE-F-006	evaluate_incremental() must accept NNUEData history array [current, ply-1, ply-2] for differential evaluation	Functional	Complete
NNUE-F-007	NNUEData must contain 64-byte aligned Accumulator with 2×256 int16 accumulations and computed_accumulation flag	Functional	Complete
NNUE-F-008	DirtyPiece must track moved pieces with dirty_num count and arrays for piece codes, from squares, to squares (max 3 entries)	Functional	Complete
NNUE-F-009	CPiece enum must define piece codes: WKing=1, WQueen=2, WRook=3, WBishop=4, WKnight=5, WPawn=6, BKing=7…BPawn=12, Blank=0	Functional	Complete
NNUE-F-010	CColor enum must define colors: White=0, Black=1	Functional	Complete
NNUE-F-011	All evaluation methods must return Err if library not initialized via init()	Functional	Complete

NNUE-F-012	FFI interface must handle CString conversions safely and validate array length consistency	Functional	Complete
NNUE-F-013	Build script must set RPATH to embed library path for runtime loading without LD_LIBRARY_PATH environment variable	Functional	Complete
NNUE-F-014	Build script must link libnnueprobe.so dynamically from assets/nnue-probe/src with absolute path resolution	Functional	Complete

6.3 · Non-Functional Requirements

ID	Description	Type	Status
NNUE-NF-001	Accumulator struct must use repr(C, align(64)) for FFI compatibility and alignment guarantees	Non-Functional	Complete
NNUE-NF-002	All FFI structs (NNUEData, Accumulator, DirtyPiece) must use repr(C) for memory layout compatibility	Non-Functional	Complete
NNUE-NF-003	Incremental evaluation must support 3-ply history to enable efficient differential updates	Non-Functional	Complete
NNUE-NF-004	Build script must invalidate cache when library or source files change via cargo:rerun-if-changed directives	Non-Functional	Complete
NNUE-NF-005	NNUEProbe and NNUEData must implement Default trait for convenient initialization	Non-Functional	Complete

6.4 · Dependencies

- `std::ffi::CString` - C string conversion for FFI
- `std::os::raw::{c_char, c_int}` - C type mappings for FFI compatibility
- `std::sync::Once` - One-time initialization synchronization
- External: `libnnueprobe.so` - Precompiled NNUE evaluation library
- Build: Rust build.rs with dynamic linking and RPATH configuration