Dodo language specification 0.1

Language specification 0.1

Edition: 15 September 2026

Status: Normative specification of the implemented 0.1 language

A small language. Explicit control. Checked borrowing.

Dodo is a small, ahead-of-time compiled systems programming language with Go-like blocks, Rust-like function signatures, checked borrowing, and explicit hardware access. It targets native applications and bare-metal firmware without requiring a garbage collector, heap, operating system, or scheduler.

Status and interpretation

This edition incorporates the September 2026 ergonomics revisions to the Dodo 0.1 design: consistent name-first declarations, shorter signatures and literals, local type inference, composable constants, value-producing blocks, ranges, immutable runtime bindings, implicit function returns, recursive patterns, conditional destructuring, subslices, explicit copying in collection loops, canonical formatting, and leading-dot continuation. Earlier type-first declarations and fully explicit forms remain accepted for compatibility. The ownership categories are unchanged.

Must, must not, shall, and shall not express requirements. May expresses permission. Should expresses a recommendation. Implementation-defined means an implementation must document its choice for each supported target; a program may depend on that documented choice, but may then require changes on another implementation or target. Appendix C records these choices for compiler 0.1.4. Unspecified means a choice need not be documented and may vary between evaluations. Undefined behavior means a program has violated a runtime validity requirement and this specification imposes no requirements on that execution. An unsafe precondition violation is not a required diagnostic or trap. A required rejection is a compile-time error; a required trap is a non-returning runtime failure (section 17).

This edition selects the implemented rules to retain as language requirements, including sequencing, representation, raw-memory contracts, C interfaces, and local package resolution. Sections carrying rule IDs are connected to acceptance, rejection, execution, or target-emission tests in Appendix B. Tests witness specific consequences of the rules; the prose defines the contract.

The specification does not assert that an implementation is complete, sound, memory-safe, mature, or secure. Examples explain the design; they are not evidence of compiler conformance. Compiler implementation status belongs in separate project documentation.

The core language rules below are normative. Library design guidance for DMA and interrupts does not imply compiler support for those facilities. Unsupported features are identified in Appendix C; they are not alternative meanings of accepted programs. The partial grammar is a reading aid, subordinate to the rules in the main text. Library APIs outside the core memory surface are versioned separately in the standard-library reference.

Contents

1. Language overview

1.1. Design profile

Dodo is an ahead-of-time compiled systems language with explicit control flow and checked ownership and borrowing. Ordinary Dodo code shall not require a garbage collector, heap allocator, operating system, or language-level scheduler.

The language supports a design for both hosted native applications and freestanding firmware. Allocation, containers, device drivers, synchronization, and scheduling belong in libraries, rather than mandatory runtime services.

1.2. Primary rules

- Mutation is explicit. Ordinary local bindings are mutable unless their declaration form constrains them.
- Aggregates have a single owner by default. Non-copy aggregates move; they do not implicitly deep-copy.
- Checked references are non-null and statically borrow-checked.
- Overlapping memory permits multiple shared readers or one mutable writer, never both while those accesses are live.
- Errors are ordinary Result values. A call site marks propagation with ? or unwraps with ! to panic on error.
- Explicit unsafe syntax contains unchecked operations.
- Explicit core-library primitives provide hardware access; ordinary mutable references must not be fabricated for device memory.
- Cleanup is deterministic on normal scope exits. Traps abort without guaranteed unwinding or cleanup.

1.3. Deliberate omissions

Version 0.1 has no operator or function overloading, inheritance, implicit constructors, exceptions, macros, reflection, closures, async/await, or language-level threads. It has no user-defined implicit conversions, iterator protocol, specialization, or metaprogramming facilities.

Operators are compiler-defined for primitive types and supported simple enums. Custom behavior uses named methods.

2. Source files, packages, and visibility

2.1. Packages

A source unit declares its package:

```
package samples
```

PKG-UNIT. Every source file must start with a package declaration, apart from whitespace, comments, and empty semicolon statements. Without a manifest, a project is a folder with a main.dodo entry file. With no input, dodo run, dodo check, and dodo build select the current folder; compile remains an alias for build. An explicit CLI directory input selects that folder. An optional

dodo.toml in the selected folder can choose a named target's entry file; otherwise the entry is main.dodo. A missing entry is an error, without searching parents or alternative filenames. An explicit file input bypasses manifests and includes exactly that file and its imports; other root files are not implicitly included. Manifest selection does not change source import resolution. No package manager, lockfile, or separate library project type is required.

An imported directory includes its immediate regular files with extension .dodo, sorted by canonical path; it must contain at least one such file. Subdirectories and symlink entries are not scanned as source files. All included files must declare the same package. Their declarations and import aliases share one package scope, independent of declaration order. Duplicate declarations are errors. Imported folders need no entry filename or special library layout. The editor may also load a directory package directly without selecting main.dodo.

2.2. Imports

Packages are imported by string path. Imported declarations use qualified names, such as mmio.write32:

```
import "core/mmio"
```

PKG-RESOLVE. An import must be a nonempty relative path of nonempty /-separated components, with no . or .. component or backslash. For a local import p, resolution from the importing unit's directory considers the file obtained by replacing p's extension with .dodo, and the directory p. Exactly one must exist; both, neither, an empty directory package, or an import cycle is an error. Thus ordinary extensionless imports use p.dodo or p/. The imported declaration must match p's final component. Package identity is its canonical resolved path, not its declared name; repeated dependencies on the same unit are loaded once. Filesystem case handling and canonicalization are implementation-defined (ID-FS).

PKG-NAMES. The final component is the default local name. import "p" as name selects an alias for the importing package, without renaming the dependency or changing its visibility. Distinct dependencies must not share a local name or conflict with the importing package's name. Repeated identical imports are permitted. A path must not have conflicting explicit aliases across files; an explicit alias also applies to unaliased repetitions of that path in the same unit. Only direct imports are in scope; imports are not re-exports.

PKG-CORE. Imports starting with core/, alloc/, or std/ must resolve from the compiler's bundled library, never local files. Unknown bundled imports are errors. core/mem, core/ptr, and core/mmio are compiler intrinsics and must be directly imported to use their calls. core.drop, core.wrapping_add, core.wrapping_sub, and core.wrapping_mul are available without an import. Package names core, mem, ptr, and mmio are reserved. These aliases are also reserved except for a matching core/mem, core/ptr, or core/mmio import. A bundled package cannot depend on a local package. Target selection for bundled platform adapters is implementation-defined (ID-LIB).

2.3. Visibility

PKG-VIS. Declarations and fields are package-private unless marked pub. Letter case has no visibility meaning. A public API must not expose a private type, except that a public method of a private struct may mention its own struct type in parameters or its return type to participate in generic protocols. A public enum exposes its variants.

Wildcard imports and re-export declarations are not supported.

PKG-INIT. There is no implicit package initialization hook or import-time execution. Package const and static initializers must be constant expressions; ordinary function calls in them are rejected.

Constant dependencies may refer forward across the loaded package graph; cycles and dependencies on mutable statics are errors. Static storage is initialized before entry to the program. This imposes no runtime order among imports, because their initialization has no side effects. A function named init is an ordinary function and is never called implicitly. Static storage lasts for the program; its contents are not automatically destroyed at program exit.

3. Lexical structure

3.1. Statements and blocks

Newlines normally terminate statements. Semicolons separate the clauses of a three-part for statement. Blocks require braces; conditions do not require parentheses. Delimited expressions and expressions continued after an operator accept newlines. A newline before . continues the preceding expression with a field or method access. Indentation, blank lines, and line comments do not affect this rule; an explicit semicolon ends the expression. Other leading operators do not continue a statement outside delimited expressions.

```
value := source
    .decode()
    .validate()
```

```
for i := 0; i < n; i += 1 {
    // body
}
```

3.2. Comments

LEX-TEXT. Source must be valid UTF-8. Space, tab, and carriage return are whitespace; line feed is the newline token, so CRLF has the same statement effect as LF. // begins a comment through the next LF or end of file. Its terminating newline is preserved. /// is an ordinary line comment. Block/nested comments, raw identifiers, raw strings, character literals without b, and a source BOM are not supported and must be rejected where used as syntax.

3.3. Identifiers and keywords

Identifiers match [A-Za-z_][A-Za-z0-9_]* and are case-sensitive. No Unicode normalization is performed on strings or comments. The following words are reserved and must not be used as identifiers:

```
package import pub fn struct enum const let return if else for in
break continue match unsafe extern as from static void mut true false
```

Self, self, primitive type names, and constructor names such as some have contextual meanings; they are not in the reserved-word set. Symbol tokens use the longest recognized spelling, including ..= before .., and >>= before >>. There is no preprocessor.

3.4. Literals

- Integer literals are constrained by context. An unconstrained integer literal defaults to isize.

- Typed integer literals include forms such as 0u32.

- Byte literals include forms such as b'a'.

- String literals, such as "hi", refer to immutable program-lifetime storage.

- Byte strings, such as b"hi", also refer to immutable program-lifetime storage.

- Array literals use [1, 2, 3], with the length inferred from the elements.

- Repeated array literals use [value; N], where N is an integer constant expression.

LEX-LITERAL. Integers use decimal or lowercase 0x, 0o, 0b prefixes for bases 16, 8, 2. Hexadecimal digits may have either case. Underscores are ignored within digit runs (including after a base prefix and at the end); at least one digit is required. The magnitude must fit u64 before contextual checking. An immediately following integer type name is a suffix. - is a separate unary operator; the magnitude of a signed minimum literal is allowed under negation.

Floating literals are decimal digit runs with a fractional part, an e/E exponent with optional sign, or an f32/f64 suffix. A fractional point must be followed by a digit; exponents require a digit. Digit separators are ignored. Floating syntax cannot take an integer suffix. The default type is f64; the text is converted to finite binary64 and then to the contextual/suffixed float type; a literal that becomes infinite in that type must be rejected. Source spellings for infinity and NaN are not supplied. true and false are Boolean literals; integers do not implicitly convert to bool.

String and byte literals accept \n, \r, \t, \0, \\, \", \', and \xHH (exactly two hexadecimal digits). Only strings additionally accept \u{H...} with one to six digits denoting a Unicode scalar value. A string's decoded bytes must be valid UTF-8. Byte literals/strings require ASCII source characters; \xHH can supply any byte. A b'...' literal must decode to exactly one byte. Literal text must not contain a physical CR or LF. Other escapes and unterminated literals are errors.

4. Types and values

4.1. Built-in types

Type	Meaning
bool	Boolean value
i8, i16, i32, i64	Signed fixed-width integers
u8, u16, u32, u64	Unsigned fixed-width integers
isize, usize	Pointer-sized signed and unsigned integers
f32, f64	Floating-point values
void	No return value
[N]T	Owned array of exactly N elements of T
&[T]	Shared slice
&mut [T]	Mutable slice
&T	Checked, non-null shared reference
&mut T	Checked, non-null mutable reference
*const T	Unchecked, nullable raw pointer for immutable access
*mut T	Unchecked, nullable raw pointer for mutable access
&str	UTF-8 string view
Result<T, E>	Success or error value

Option<T>	Optional value, conceptually some(T) or none

4.2. Arrays and slices

An array [N]T owns exactly N elements. N is a nonnegative integer constant expression, including references to named constants. Array literals infer their length and take their element type from context, typed elements, or the ordinary literal defaults:

```
values: [4]u16 = [1, 2, 3, 4]
bytes := [0u8; 256]
```

[value; N] evaluates value exactly once, including when N is zero, and repeats it N times. The element must be copyable; repetition never clones owned values or duplicates mutable references. Empty lists need an element type from context. Bracket lists and repetition are canonical. A list may also carry an explicit type annotation in expression position: ([1, 2, 3, 4]: [4]u16) or ([]: [0]u8). This annotation checks the element type and exact length without converting or copying an existing aggregate. It applies only to an unannotated bracket list; annotate the binding when using repetition. The historical composite form [4]u16{1, 2, 3, 4} remains accepted and is automatically migrated to an annotated bracket list by dodo fmt.

Arrays and slices expose .len. Borrowing an array may coerce to a slice without allocation. &values[start..end] produces a shared subslice and &mut values[start..end] produces an exclusive mutable subslice. A bare values[start..end] also produces a shared subslice. Missing start and end bounds mean zero and the collection length respectively. The source, start, and end are evaluated once, in that order. Bounds must satisfy 0 <= start <= end <= length; violations trap in every optimization profile. An empty slice at the end of a collection is valid. Bounds may read the captured source (for example, data.len) but must not move or mutate it; mutable access is established after the bounds have been evaluated.

Subslices retain their source's checked lifetime. Mutable slicing requires exclusive access; ranges do not establish disjointness between simultaneous mutable slices. UTF-8 string slicing is not provided by this syntax.

4.3. Checked references

&T permits shared reading. &mut T permits mutation and grants exclusive access to the referenced region for the loan's duration. Both are checked and non-null.

Field access and indexing automatically dereference checked references. Raw pointers are never automatically dereferenced.

4.4. Raw pointers

*const T and *mut T are unchecked and nullable. Copying or storing a raw pointer does not dereference memory. Raw-pointer dereference, pointer arithmetic, and conversion from a raw pointer to a checked reference require unsafe.

4.5. Structs

Structs contain named fields and use named-field literals. Fields remain package-private unless individually marked pub.

```
pub struct Pin {
    set_address: usize
    clear_address: usize
    mask: u32
}

pin := Pin{
    set_address: set,
    clear_address: clear,
    mask,
}
```

A literal field written as mask is shorthand for mask: mask. It reads or moves the binding exactly as the explicit initializer does.

4.6. Enums

Enums are plain tagged values or tagged values with payloads. A public enum exposes its variants. Matching may destructure payloads.

```
pub enum ParseError { Length, Digit }

enum Token { End, Byte(value: u8), Pair(u8, u8) }
```

REP-TAG. Variants may have no payload or a parenthesized ordered list of payload fields. Fields may be bare types, name: Type, or the historical Type name. Explicit discriminant expressions are rejected. Variants have zero-based consecutive u32 tags in declaration order. Option<T> uses none/none() (tag false) and some(value) (tag true); Result<T,E> uses ok(value) (tag false) and err(error) (tag true). ok() constructs void success. Only the active variant's payload is a live value and participates in ownership and destruction. Invalid tags and invalid active payloads violate runtime validity. Section 16.2 fixes their storage structure.

4.7. Generics

Basic type-parameter generics use angle brackets, as in Name<T>. Every instantiation must satisfy ordinary type and ownership checks. There are no traits, specialization, user-defined implicit conversions, or metaprogramming.

Function calls and struct literals infer omitted type arguments from their arguments or fields and an expected result type. For example, identity(42i32) and Box{value: 42i32} infer i32. An expected type also constrains unsuffixed literals, as in value: i32 = identity(42). Inference is local; it does not infer function signatures from bodies or solve types from later statements. Conflicting or unresolved type arguments require a diagnostic. Explicit arguments remain available, canonically as identity::<i32>(42). The historical call spelling identity<i32>(42) remains accepted and is migrated by dodo fmt. Generic type names and declarations continue to use Name<T>.

some(value) infers Option<T> from its payload when no expected type exists. none, ok, and err still need enough context to determine their full types. GEN-INSTANCE. Generic calls and types are instantiated with concrete types in the loaded program; each used instantiation must pass ordinary semantic checks. There are no generic constraints, variance annotations, subtyping between different instantiations, or separately compiled generic interfaces. Implementations may share generated code only when observable behavior is preserved. Expansion limits must be diagnosed and documented (ID-LIMIT).

4.8. Scalar and view representations

REP-SCALAR. Integers have exactly their named bit widths; signed integers use two's complement and unsigned integers use binary representation with no invalid bit patterns. isize and usize have the selected target's default data-pointer width, not the compiler host's width. A byte is eight bits. bool has values false and true, stored as one byte, respectively 0 and 1; other byte representations do not establish a valid Boolean value. f32 and f64 use IEEE 754 binary32 and binary64 encodings, including signed zero, infinities, and NaNs. Integer/float alignment and byte order are target properties (ID-TARGET).

REP-VIEW. A checked reference or raw pointer occupies one default data-pointer slot. A slice or &str occupies an ordered pair (data pointer, usize length) with target struct padding/alignment. Slice length counts elements; string length counts UTF-8 bytes. &str indexing is rejected. Literal strings and byte strings have immutable storage lasting for the program. Their length excludes any bytes outside the literal, and no trailing NUL is guaranteed. Literal pooling and literal addresses, padding bytes, inactive payload bytes, and NaN payload/sign selection are unspecified; programs must not use them as value identity or serialization. Only the explicitly initialized bytes of a value may be read as initialized bytes. Copying an object does not guarantee preservation of padding.

5. Bindings and initialization

5.1. Local bindings

:= declares a mutable local whose type follows from its initializer and surrounding constraints. Explicit declarations use name: Type, consistently with parameters, fields, and enum payloads:

```
name := value
count: u32 = 0
```

Name-first declarations are canonical, including fields, constants, statics, and named enum payloads. dodo fmt migrates earlier type-first declarations, array literals, and explicit generic calls, while preserving explicit type information. Historical forms remain accepted in 0.1; a future revision may announce their deprecation after the migration path is established.

let declares an immutable runtime binding. Its initializer runs normally and may call functions; a type annotation is optional. Initialization is mandatory:

```
let limit = read_limit()
let typed_limit: u32 = read_limit()
```

An immutable binding cannot be reassigned, mutably borrowed, or used to mutate its owned fields or array elements. Immutability does not change the type or permissions of a reference it holds: let r = &mut value allows *r = next and writes to the referent's fields. An immutable binding holding a mutable slice likewise permits element writes. Replacing r itself is forbidden. Moving an owned value out of an immutable binding remains allowed under the ordinary move rules. Runtime let bindings cannot be used as compile-time constants, even when their initializers are literals.

5.2. Constants

const declares a compile-time constant:

```
const LIMIT: u32 = 64
```

Constants may refer to other constants, including imported public constants and previously declared local constants. Package constants may refer forward; cycles are rejected. Arithmetic uses the declared types and selected target width. Runtime variables and mutable static storage cannot determine a constant or array length. Compile-time function calls are not part of this revision.

5.3. Definite initialization

Every read requires definite initialization. A moved-from binding is not initialized for reading until it is reinitialized. The implementation must reject a read unless the binding or subobject is established as initialized on every applicable control-flow path.

6. Functions and methods

6.1. Functions

An omitted return type means void; fn reset() {} and fn reset() -> void {} have the same signature. Functions returning a value must declare its type. A function returning a value returns its final expression when that expression has no trailing semicolon. This applies recursively to final conditionals, matches, and blocks; every continuing path must produce the declared return type. return remains available anywhere for early exits. Void functions retain statement semantics and reject discarded Results.

```
pub fn add(a: u32, b: u32) -> u32 {
    a + b
}
```

6.2. Arguments

Passing a copy type copies its value. Passing an owned non-copy value transfers ownership. Passing a checked reference passes or reborrows access according to the borrowing rules. Free-function calls make new borrows explicit:

```
update(&mut item)
```

Reference arguments are reborrowed when needed.

6.3. Methods and associated functions

Methods are declared inside their struct. Their first parameter is named self:

Receiver	Effect
&self	Shared read access
&mut self	Exclusive mutable access
self	Consumes the value

The equivalent explicit forms self: &Self, self: &mut Self, and self: Self remain accepted. Shorthand is valid only for struct receivers.

item.update() automatically borrows or reborrows the receiver as needed. A function inside a struct without a receiver is called as Type.name(...). Dispatch is static. There is no inheritance or separate implementation block.

6.4. Borrowed returns

A function returning a checked borrow must identify the source that bounds the result's lifetime:

1. A borrowed receiver is the inferred source when present.

2. Otherwise, exactly one borrow-carrying parameter is the inferred source.

3. An ambiguous signature declares its sources with from(...).

4. A borrow with no input source requires from(static) and program-lifetime storage.

```
fn choose(a: &u8, b: &u8, first: bool) -> &u8 from(a, b) {
    // body
}
```

The result conservatively keeps every named source borrowed. The compiler must check the function body against its declared or inferred contract. A contract does not extend a lifetime and never permits a reference to a local destroyed on return. Source-based contracts replace user-written lifetime parameters, not lifetime checking.

6.5. Compiler-checked printing

The std/console and std/fmt printing entry points are a narrow exception to ordinary fixed-arity calls. print(value) and println(value) select primitive formatting automatically and shared-borrow custom value places. printf takes a literal format string and heterogeneous arguments; the compiler checks its placeholders, argument count, and formatting compatibility. The portable fmt forms take an explicit mutable writer first. console.Output supplies equivalent methods. All forms return usize!io.Error and preserve ordinary Result handling.

{} consumes the next argument; {{ and }} escape braces. Initial options and type compatibility are specified in the [formatting reference](formatting.md#checked-format-strings). Receiver/writer and argument expressions evaluate once, left to right, before output. Custom temporaries are owned during formatting and cleaned up on normal success or error. Custom values use a public shared format method returning void!io.Error; the existing structural formatter contract remains valid.

These calls lower to ordinary statically typed functions and checked borrows. One formatter records cumulative progress across the whole call, including a failing write's partial progress. No C variadics, runtime format parsing, or allocation are introduced. The bodyless @compiler(print), @compiler(println), and @compiler(printf) declarations are reserved for the bundled std/fmt and std/console packages; user packages cannot declare them. This facility does not provide general argument packs or function overloading.

7. Ownership and moves

7.1. Copy categories

Category	Types
Copy by default	Scalars, shared references, raw pointers
Move by default	Structs, arrays, tagged values, mutable references

Passing or assigning an owned move value transfers ownership. A moved-from binding cannot be read until reinitialized. Duplicating an aggregate requires an explicit ordinary function, such as

clone; there is no implicit deep copy.

7.2. Subobjects

Separate struct fields may be borrowed independently. Potentially overlapping indexed accesses are rejected unless a checked operation, such as slice splitting, establishes disjointness.

OWN-SUBOBJECT. Moving a non-copy field or indexed element as a standalone expression is rejected. Move the whole owner or use the destructuring patterns in section 11.3, which consume the owner and destroy uncaptured values. Those patterns must not move fields out of a value with a custom drop method. There is no general partially moved aggregate state in 0.1.

8. Borrowing and reference validity

8.1. Aliasing

For overlapping memory, a program may have multiple live shared readers or one live mutable writer, never both. While a loan is live, its owner cannot move, be destroyed, or access the borrowed region incompatibly. Safe code cannot convert a shared reference to a mutable reference.

8.2. Loan extent

Borrowing is checked at compile time; it does not require a runtime reference counter. A loan ends after its last possible use on each control-flow path, including any destructor that could access it. Consequently, a loan can end before its enclosing lexical block if no later path can use it.

8.3. Reborrowing

Reborrowing a mutable reference temporarily suspends the original reference for the duration of the reborrow. It does not duplicate exclusive access.

8.4. Borrow-carrying values

Structs may contain references. A value carrying borrows has one conservative inferred lifetime bounded by every retained source. Result, Option, and containers propagate this dependency.

Replacing a borrowed field cannot extend the value's inferred lifetime. Independently varying field lifetimes and self-referential owning structs are excluded from version 0.1.

8.5. Disjoint indexing

Potentially overlapping indexed accesses are rejected unless a checked operation establishes disjointness. The language shall not silently add copying, allocation, or runtime borrow checks to accept a conflict.

8.6. Safety scope

Memory safety is a goal for safe code, conditional on a sound compiler, a sound core library, and correct unsafe and FFI contracts. It is not a general security guarantee.

9. Destruction and cleanup

9.1. Normal exits and replacement

Owned values are destroyed in reverse declaration order on ordinary scope exits, including return, ?, break, and continue. Moved values are not destroyed twice. Overwriting an initialized owned value destroys the old value before replacing it.

9.2. Custom destruction

A struct may declare the reserved method:

```
fn drop(&mut self) {
    // cleanup
}
```

The method runs before the fields are destroyed. Struct fields and enum payload fields are destroyed in reverse declaration order; array elements are destroyed in decreasing index order. Only active tagged payloads are destroyed. It cannot return an error or be called directly. core.drop(value) consumes and destroys a value early. Moving fields out of a value with a custom drop method is forbidden.

9.3. Traps

Traps abort without unwinding or guaranteed cleanup. Programs must not depend on cleanup running in response to a trap.

10. Expressions and conversions

10.1. Access and calls

Operation	Syntax
Function call	f(args)
Method call	receiver.method(args)
Field access	value.field
Indexing	value[index]
Shared borrow	&value
Mutable borrow	&mut value
Explicit dereference	*value

Field access and indexing automatically dereference checked references, never raw pointers. An explicit dereference obtains a scalar value where needed.

10.2. Numeric conversion

NUM-CAST. as between integers must trap if the source value is outside the destination's range. It does not truncate bits, wrap, or saturate. Integer-to-float conversion rounds directly to the destination representation, with identical results in constant expressions and at runtime. Identity floating casts and widening f32 to f64 preserve the numerical value, including NaN, infinities, and

signed zero, in both constant expressions and at runtime. Narrowing f64 to f32 requires a finite source in [-f32::MAX, f32::MAX] (where MAX denotes the largest finite binary32 value), then rounds; NaN and infinity must trap on this narrowing conversion.

Float-to-integer conversion first requires a finite source x in [-2^(N-1), 2^(N-1)) for signed N-bit integers or [0, 2^N) for unsigned integers, and then truncates toward zero. The test is on x before truncation: -0.5 as u8 traps, 255.75 as u8 is 255. Conversion failure in an evaluated constant expression is a compile-time error. Floating constant precision is implementation-defined (ID-FLOAT). Raw-pointer casts have the separate rules in section 14.2; Boolean and enum numeric casts are not supported.

```
weight := i as u32 + 1
total += weight * (value as u32)
```

10.3. Operators

EXPR-PREC. Operators are compiler-defined and cannot be overloaded. This table lists precedence from weakest to strongest. Binary operators and repeated casts associate left; prefix operators nest right; postfix operations chain left.

Level	Operators
1	||
2	&&
3	|
4	^
5	&
6	==, !=
7	<, <=, >, >=
8	<<, >>
9	+, -
10	*, /, %
11	as Type
12	Prefix -, !, ~, *, &, &mut
13	Calls, .field, [index], subslices, postfix ? and !

Parentheses override precedence. = and the compound assignments +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>= are statements, not expressions. Ranges are restricted to loop, slice, and pattern syntax; they are not binary operators on general values. Comparison chains do not have special semantics: a < b < c means (a < b) < c and must type-check that way.

Operands must have the same type after contextual literal inference. Already typed operands never undergo implicit width or signedness conversion. Arithmetic supports integers and floats; bitwise operators and shifts require integers. Boolean &&, ||, and ! require bool; equality supports bool, numeric values, raw pointers, and payload-free enums. Ordered comparisons support numbers only. No aggregate equality is implied.

EXPR-ORDER. Evaluation is sequenced left to right: complete each operand's effects before starting the next. Calls evaluate the receiver (if any), then arguments in source order, then enter the callee. Array elements, enum payload arguments, and struct field initializers evaluate in their written order; struct declaration order does not reorder initializer effects. An index evaluates its collection before its index. A slice evaluates its collection, start, then end; a range evaluates start then end. Each is evaluated once. Repetition evaluates its initializer exactly once even for length zero (section 4.2).

a && b evaluates b only when a is true; a || b evaluates b only when a is false. if evaluates only the selected branch. Match selection follows section 11.3. ?, return, break, continue, or a trap stops evaluation of later operands on that path. Temporaries already owning values must receive normal cleanup on non-trapping exits, in reverse creation order. Moving into a completed call or aggregate transfers that cleanup obligation. A returned/yielded value transfers before scope cleanup, including a void call's effects.

EXPR-ASSIGN. place = rhs evaluates the destination address once (including its base and indices), then rhs, then destroys any initialized old owned value, then stores the replacement. If rhs exits early, the old value remains owned and receives ordinary exit cleanup. place op= rhs evaluates that address once, loads the old value, evaluates rhs, applies the checked operator to the captured old value and rhs, then stores. It does not re-evaluate the place or reload its old value after rhs. All steps remain subject to borrow checking.

NUM-ARITH. Integer +, -, *, unary negation, and left shift must trap if the mathematical result is unrepresentable. Integer division truncates toward zero; remainder has the dividend's sign or is zero. Division and remainder both trap for a zero divisor and for signed minimum divided by -1. Shift counts must be nonnegative and less than the left operand's width. Right shift sign-extends signed values and zero-extends unsigned values. Bitwise operations act on the fixed-width representation. Evaluated constant failures must be diagnosed. core.wrapping_add(a, b), core.wrapping_sub(a, b), and core.wrapping_mul(a, b) explicitly wrap modulo 2^N, where N is the operand type's width on the selected target. Both operands and the result must have the same integer type. These safe calls accept two value arguments and an optional explicit integer type argument; otherwise the type is inferred from the expected result or operands, defaulting to isize for unsuffixed literals. Signed results interpret the low N bits as two's complement. The wrapping operation never traps for overflow; its argument expressions retain their usual checked behavior. These calls are not constant expressions.

NUM-FLOAT. Floating arithmetic uses IEEE operations without reassociation or fast-math assumptions. Division by zero and overflow may produce infinities or NaNs; they do not invoke the integer traps. % is remainder using a quotient truncated toward zero. Equality considers the two signed zeros equal. With a NaN operand, != is true and all other comparisons are false. Runtime operations and integer conversions assume the target's default floating environment; rounding, subnormal support, and constant evaluation are documented under ID-FLOAT. Foreign code must restore that environment before executing Dodo code.

11. Statements and control flow

11.1. Blocks and conditions

A block is a brace-delimited sequence of statements. Control-flow constructs that take blocks require braces. Conditions must have type bool; numeric truthiness is rejected. Parentheses are optional around conditions.

```
if condition {
    // ...
} else if other {
    // ...
} else {
    // ...
}
```

if, match, unsafe, and plain blocks can also produce values in expression positions. The final expression in a value block supplies its value:

```
limit := if fast { 100u32 } else { 10u32 }
value := unsafe { ptr.read(pointer) }
```

Every continuing path must supply a value of the same type, and at least one value-producing path must exist. Other paths may return, propagate an error, break, or continue as permitted by their enclosing function and loops. Only the selected branches execute. A value is transferred out before the block's locals are destroyed in reverse order. Borrows of the block's local storage cannot escape. Newlines and semicolons remain statement separators inside value blocks.

11.2. Loops

for is the only loop keyword. break and continue apply to the nearest enclosing loop.

```
for { ... }
for condition { ... }
for i := 0; i < n; i += 1 { ... }
for i in 0..n { ... }
for _ in 0..n { ... }
for item in items { ... }
for index, item in items { ... }
for &item in items { ... }
for index, &item in items { ... }
for item in &mut items { ... }
for index, item in &mut items { ... }
```

A range loop for i in start..end visits integers from start up to, excluding, end. Its integer bounds have the same type, inferred from typed bounds or the ordinary integer default. Both bounds are evaluated once, left to right, before the loop. A reversed or empty range performs zero iterations. Each iteration gets a fresh binding; assigning to it does not alter the loop counter. _ discards the iteration value. break and continue keep their ordinary cleanup semantics. Ranges are built-in loop syntax, without an iterator protocol or a first-class range type.

Collection foreach supports arrays and slices. It evaluates its input once, borrows rather than consumes the collection, and creates fresh bindings on each iteration. Indexed forms bind a usize index. Shared iteration binds &T elements; mutable iteration binds &mut T elements. These meanings never depend on the element type.

A shared reference pattern opts into copying: for &item in items or for index, &item in items binds a fresh value of type T on each iteration. T must be copyable under section 7; owned aggregates and mutable references cannot be copied. Reassigning item only changes the local copy. Copying a shared reference preserves its dependency on the original storage. The collection is still evaluated once and borrowed for the loop. The pattern applies only to shared collection elements; reference patterns on range values, indices, or mutable iteration are rejected. Use a shared reborrow to copy from a mutable view. &mut item binding patterns are not supported.

The collection stays borrowed for the iteration. A mutable element loan cannot escape its iteration in version 0.1. There is no iterator protocol and no while, loop, or do keyword.

11.3. Matching

match is exhaustive and has no fallthrough. Arms accept single expressions or braced blocks in both statement and expression positions:

```
match result {
    ok(value) => consume(value),
    err(error) => report(error),
}
```

Blocks allow multiple statements. When the match produces a value, each continuing arm yields its final expression:

```
return match parsed {
    ok(value) => value,
    err(_) => fallback,
}
```

The same recursive pattern grammar is used by match, if let, and let:

- _ ignores a value; a binding name captures it.
- Boolean and integer literals test equality. Integer and byte ranges use start..end (exclusive) or start..=end (inclusive); bounds must be integer literals representable in the matched type, with a nonempty ordered range.
- Enum/Option/Result payloads contain patterns, such as some(some(value)). Enum variant names may omit the qualifier when the matched type identifies it.
- Struct patterns use Name{field: pattern, shorthand, ..}. Without .., every field must be listed; shorthand binds the field's name.
- Alternatives use p | q and must bind the same names with the same types and borrow modes in every alternative. Alternatives may be nested in payloads.

Arms are considered in source order. An optional if condition guard runs after a pattern matches, with its bindings in scope. A false guard tries the next alternative or arm. Guards must be Boolean and cannot move or mutate the matched bindings before the arm is selected. Guarded arms do not establish exhaustiveness. These alternative and guard rules follow the [Rust match reference](https://doc.rust-lang.org/reference/expressions/match-expr.html).

Matching an owned non-copy value consumes it; matching &value or &mut value borrows its payloads recursively. Ignored owned fields are destroyed once. Destructuring cannot move fields out of a struct with custom drop.

11.4. Conditional and early-exit bindings

```
if let some(value) = optional {
    consume(value)
}

let some(value) = optional else {
    return
}
```

if let evaluates its scrutinee once and makes immutable bindings available only in the success block; an optional else handles failure. let patterns introduce immutable bindings in the enclosing scope. A refutable let pattern requires an else block that exits the current path, for example with return, break, continue, or an infinite loop. An irrefutable pattern needs no else. The failure block cannot access the new bindings. Owned scrutinees are consumed on either path, with unused values destroyed; borrowed scrutinees remain borrowed.

Conditional destructuring must cover every path that carries a Result. Success-only ok(value) patterns are rejected; use an exhaustive match with explicit ok and err handling, or propagate with ? first. This restriction also applies to nested Results and borrowed Results. some(result) may match an Option<Result<T, E>> because the unmatched none path carries no Result, but the captured result must still be handled in the success block. Patterns cannot discard an unhandled Result in a wildcard, omitted field, or payload. A statement arm that produces a new Result must still forward, propagate, or handle it.

12. Results and error propagation

12.1. Result values

T!E is shorthand for Result<T, E>. It is not an exception type and does not imply a separate ABI. ! binds outside reference and slice type syntax.

The prelude supplies ok(value) and err(error). A fallible function must return one of these explicitly. A void!E function succeeds with ok().

12.2. Propagation

A trailing ? on a call result produces the success value or immediately returns the same error from the enclosing function:

```
high := nibble(text[0])?
low := nibble(text[1])?
```

? is permitted only when the enclosing function returns a Result with the same error type. It performs no implicit conversion. Cleanup follows the normal return rules.

12.3. Local recovery

match handles failure locally. There are no exception classes, error traits, implicit error conversions, or configurable propagation operators. Error translation explicitly matches a source error and constructs a destination error.

12.4. Mandatory handling

Discarding a Result, including through _ =, is a compile-time error. A program must forward it, propagate it, unwrap it with !, or explicitly handle both variants with match. An intentionally empty error arm counts as handling.

12.5. Unwrap or panic

A postfix ! evaluates an owned Result<T, E> once and produces its ok payload of type T. An err triggers a result unwrap panic at the expression's source location using the selected panic strategy (ID-TRAP). Panic does not unwind or run destructors. Unlike ?, ! works in functions with any return type and does not return an error to the caller.

The Result is consumed. On success, its payload retains ownership and borrow dependencies and is destroyed normally when its owner leaves scope. void success produces no value; a nested Result still requires handling. ! has the same precedence as postfix ? and chains with field access, indexing, and calls. Field access and indexing on an owned aggregate payload require binding it to a local first; reference and slice payloads can be accessed directly. Prefix ! remains Boolean negation, and ! in type syntax remains Result shorthand.

13. Unsafe code

13.1. Unsafe blocks

unsafe introduces a lexical block permitting particular unchecked operations:

```
unsafe {
    mmio.write32(self.set_address, self.mask)
}
```

The following operations require unsafe context:

- Raw-pointer dereference and arithmetic.

- Construction of a checked reference using an owner-bound raw-pointer primitive.

- Foreign function calls.

- Unchecked memory access.

- Target-specific unchecked operations where provided by an implementation.

An unsafe block does not disable type checking, ordinary borrowing rules, or safe indexing checks.

13.2. Unsafe functions and wrappers

An unsafe function places safety obligations on its caller:

```
pub unsafe fn claim(set: usize, clear: usize, mask: u32) -> Pin {
    // ...
}
```

Its body still requires explicit unsafe blocks around unchecked operations. Public unsafe functions must document their preconditions. Each unsafe block should explain why those preconditions hold at the operation site.

A safe wrapper must enforce the underlying unsafe preconditions for every safe caller allowed by its public API, not merely one expected caller.

14. Raw-memory validity

14.1. Allocation identity and permitted access

PTR-VALID. An allocation is one contiguous storage region with an extent, alignment, and lifetime. A local object's storage, a static object, and a region returned by a foreign allocator each have an allocation identity. Fields and elements are subobjects of their enclosing allocation. Moving a value transfers ownership, not an assurance that its address stays fixed. Ending a local storage lifetime or freeing/reallocating foreign storage invalidates pointers into that storage; reuse of its numerical address does not revive its identity.

A usable raw pointer consists conceptually of an address and authority to access an originating allocation. The implementation need not store that authority in pointer bits or track it dynamically. Every nonempty access must stay within one live allocation, with permission to read or write all accessed bytes. Typed loads/stores require the type's alignment, except explicit unaligned operations. Reads require initialized, valid values (including tags and active payloads). Writes must establish a valid value before a later typed read. Creating checked references to uninitialized T is invalid; MaybeUninit<T> supplies separate storage without asserting T's initialization.

Raw access may reinterpret initialized bytes as another type if extent, alignment, value validity, and ownership requirements all hold; there is no additional effective-type rule based on the original declaration's type. Bitwise copying an owning value must not create two owners that will both be used or destroyed. ptr.read does not tell the checker that the source was moved; ptr.write overwrites without destroying the previous value. Accounting for ownership and eventual destruction is the unsafe caller's obligation.

PTR-ALIAS. Raw-pointer copying alone does not create an exclusive loan. However, every raw access must respect live checked loans: shared-borrowed storage must not be mutated or invalidated except through the specified atomic storage primitives of section 15.4, and exclusively borrowed storage must not be accessed through an independent route while that loan is live. Access through a pointer derived from the active exclusive reference is allowed with its permissions and reborrow restrictions. Raw pointer casts cannot upgrade read-only storage or a shared loan to writable storage. Concurrent conflicting non-atomic accesses are invalid; volatile is not a synchronization exception. These are runtime obligations even where the compiler accepts the unsafe code.

14.2. Casts, addresses, and offsets

PTR-CAST. A checked &T can safely convert to *const T, and &mut T to *mut T or *const T, with the same pointee type. ptr.from_ref, ptr.from_mut, ptr.as_ptr, and ptr.as_mut_ptr provide the corresponding reference/collection conversions. Obtaining a raw pointer neither extends storage lifetime nor keeps a checked loan alive by itself. A shared reference must not convert to *mut T.

Raw-pointer-to-raw-pointer casts, integer-to-raw-pointer casts, and raw-pointer-to-integer casts require unsafe. They do not access memory or check the resulting address. On the integral-address profile in ID-PTR, integer/pointer casts preserve low bits, discard excess high bits, and zero-extend when widening, regardless of integer signedness. They are distinct from checked numeric casts. Zero converts to null; ptr.is_null tests null without accessing storage. Raw pointer equality compares addresses; it does not establish allocation identity.

Converting an allocation-derived pointer to an integer exposes that allocation's address. Converting the unchanged usize address back while the allocation is live must recover a usable pointer with the original access permissions. Address arithmetic may recover another address in that same exposed allocation, provided it does not overflow and the entire accessed range remains valid. An arbitrary number alone grants no allocation or access permission. A foreign API may supply a live allocation and its address; its allocation/free contract determines the lifetime. Access to fixed hardware addresses has the separate target contract in section 15.2. Non-integral pointer targets must document an alternative address interface or reject these operations (ID-PTR).

PTR-OFFSET. ptr.offset(p, n) takes n: isize, in units of the pointee's size, and preserves p's allocation and permissions. For allocation-derived p, both positions must lie in that allocation or one past its end; the signed byte displacement must fit isize and address calculation must not wrap. One-past pointers may be brought back inside but must not be used for a nonempty access. No dynamic bounds or overflow checks are required by this unsafe operation.

14.3. Constructing checked views

PTR-VIEW. Raw-pointer-to-checked-reference casts and &raw_pointer must be rejected, including inside unsafe. The explicit unsafe operations ptr.borrow(p, owner), ptr.borrow_mut(p, owner), ptr.borrow_slice(p, count, owner), and ptr.borrow_slice_mut(p, count, owner) are the bridge for plain opaque elements. The separately checked typed-storage operations ptr.store, ptr.take, ptr.relocate, ptr.view, and ptr.view_slice, with stores(...), from(owner.stored) and requires_plain(...) contracts, are specified in the [container element guide](container-elements.md). They require a matching typed owner witness and reject Result-bearing and exclusive-reference elements. For ptr.borrow, the owner must be a checked reference or slice; mutable views require a mutable pointer and exclusive owner. The result retains the owner's complete checked dependencies, including inherited shared dependencies. It must not outlive or conflict with the owner. Pointer, count (if present), and owner evaluate in that order, including the owner's effects even though its address need not appear in the returned view.

The unsafe caller must ensure that the owner keeps the complete referent alive and valid for the returned loan. The pointer need not be inside the owner's own representation: a box may anchor its separately allocated contents. Nonempty views must cover initialized elements in one live allocation; their byte extent must fit isize. Empty and zero-sized views still require a nonnull aligned pointer, even though no bytes need to be accessed. Element types recursively containing checked references or Results must be rejected, since this operation cannot reconstruct their hidden lifetime or handling obligations.

The compiler enforces the owner loan and type restrictions, not the pointer/owner correspondence. A cast or unsafe block cannot make an invalid view valid.

14.4. Byte transfers and destruction

PTR-BYTES. ptr.copy(src, dst, count) copies count elements' bytes with overlap permitted, as if through temporary bytes. ptr.copy_nonoverlapping requires disjoint ranges. ptr.write_bytes(dst, byte, count) fills count * size_of<T>() bytes. That product must fit usize; nonempty ranges require appropriate read/write permission. These operations require only byte alignment. Zero-byte operations permit null pointers, including nonzero counts of zero-sized elements. They do not by themselves prove initialization or create independent ownership. Typed raw access/copies/fills of checked-borrow-carrying elements must be rejected. ptr.drop_in_place requires one live, aligned, initialized T, destroys it and its fields, and does not free its storage. It must reject checked-borrow-carrying or Result-containing T. The caller must prevent subsequent use or another destruction until reinitialization.

15. Core memory and hardware primitives

15.1. Pointers and memory

CORE-MEM. core.ptr provides the operations specified in section 14, including ordinary, unaligned, and volatile read/write forms. core.mem provides layout queries, opaque storage, exchange, and UTF-8 views. The [implemented core call table](memory-and-ffi.md#implemented-core-calls) gives their normative signatures and type restrictions for this edition. Unknown intrinsic names must be rejected. mem.uninit::<T>() creates opaque MaybeUninit<T> storage; mem.assume_init is unsafe and consumes storage that the caller has fully initialized. MaybeUninit<T> must never implicitly destroy a T. mem.replace returns the old value without dropping it; mem.swap exchanges two disjoint initialized values. Both reject checked-borrow- or Result-containing types. mem.str_bytes preserves a string's borrow; unsafe mem.str_from_utf8 requires valid UTF-8 and

preserves the input slice's dependencies.

Allocation is optional and explicit. Recoverable allocation failure returns Result.

15.2. Memory-mapped I/O

core.mmio supplies width-specific volatile operations, including forms equivalent to:

```
unsafe fn read32(address: usize) -> u32
unsafe fn write32(address: usize, value: u32) -> void
```

CORE-MMIO. These operations address device memory directly without fabricating an ordinary mutable reference. Only access widths supported by the target are accepted. Unsupported widths must not silently become multiple narrower accesses. The 0.1.4 profile accepts 8, 16, 32, and 64 bits up to the target pointer width; actual device legality is a platform precondition (ID-HW).

Valid addresses, permissions, device side effects, and configuration are unsafe obligations of the caller.

15.3. Volatile operations

Volatile operations must not be removed or coalesced. They must retain compiler-level ordering with other volatile operations. They are not atomics, CPU barriers, or synchronization operations.

Interrupts, DMA, and multiple cores require appropriate atomics, target barriers, or correctly scoped critical sections.

15.4. Mutable static state

Mutable static globals require unsafe access unless a synchronization primitive encapsulates them. Core atomics are the narrow exception to ordinary immutable shared-access restrictions; they must not expose ordinary mutable references to storage accessible concurrently.

Interrupt-safe wrappers must account for every context that can reach the underlying storage.

15.5. DMA

A DMA wrapper must keep its transfer buffer alive and unavailable for conflicting CPU access until completion. Dropping an active transfer must safely stop it or retain the buffer until the device stops accessing it.

A move-only peripheral handle prevents accidental language-level duplication, but does not establish hardware exclusivity. The board or platform layer must establish that property.

16. Foreign interfaces, layout, and target attributes

16.1. C ABI

ABI-C. A C-ABI declaration uses syntax such as:

```
unsafe extern "C" fn send(data: *const u8, length: usize) -> i32
```

Only extern "C" is supported. A declaration may be a prototype without a body or a definition with a Dodo body. A call to either requires an explicit unsafe block, even without an unsafe declaration modifier. Parameters/results must be primitive integers, floats, bool, or raw pointers; void is permitted only as the result. Checked references, slices, arrays, structs (including @repr(C)), enums, Option, and Result by value are rejected. Variadic declarations are rejected. Use a raw

pointer to pass compatible aggregate storage.

The selected target's C calling convention determines parameter passing and returning. Fixed-width integers correspond to C integers of the same width and signedness, bool to C _Bool, f32/f64 to target binary32/binary64 C floating types, and raw pointers to C data pointers. usize/isize use the pointer-width integer ABI, not necessarily C unsigned long/long. Bindings must supply exact target C declarations and satisfy the foreign function's validity requirements. Narrow argument/result extension must match the target ABI (ID-C).

An external symbol is the final declared function name, independent of package aliases. Compatible repeated declarations refer to one symbol. Declarations with different lowered parameter/result types must fail before native linking; the binding author must also ensure consistent signedness and extension contracts where those types have the same storage width (ID-C). C entry points must not unwind through Dodo frames; Dodo has no exception unwinding contract.

16.2. Layout and attributes

REP-LAYOUT. Struct fields retain declaration order, with no field reordering or packing. The first field starts at offset zero; each subsequent field starts at the first offset after its predecessor satisfying its target ABI alignment. The struct's size is rounded up to its target aggregate alignment, which must accommodate all fields. Arrays store elements contiguously with stride size_of<T>(); [0]T has size zero and T's alignment. Empty structs have size zero; distinct zero-sized objects need not have distinct addresses.

@repr(C) uses this same field-order algorithm with the target's C-compatible alignment and padding. It does not make a Dodo-only field type a C type or authorize aggregate-by-value C calls. mem.size_of::<T>(), mem.align_of::<T>(), and mem.offset_of::<T>("field") must report the selected target's ABI size, alignment, and direct accessible struct-field offset. Private-field access through offset_of is rejected. The special queries for void are size zero and alignment one. MaybeUninit<T> has T's size/alignment.

Enum storage is the ordered aggregate (u32 tag, shared_payload). Each variant's payload is an ordered struct of its fields, including an empty struct for a payload-free variant. All alternatives start at the same offset within shared_payload. Its alignment is the maximum alternative alignment (at least one), and its size is the maximum alternative size rounded up to that alignment. Result storage is (bool is_error, shared_payload), with overlapping alternatives T and E under the same rules; a void alternative has size zero and alignment one. The enclosing tag/payload aggregate uses the struct alignment/padding algorithm, so padding may precede the payload and follow the aggregate. Option storage remains (bool is_some, T payload) under that algorithm.

For example, on Cortex-M0, an enum with two [256]u8 alternatives has size 260 and alignment 4, and Result<[256]u8, [256]u8> has size 257 and alignment 1. This shared representation replaces the earlier separate variant slots and void placeholder; objects compiled with the earlier layout must be rebuilt together. The existing tag values are unchanged (REP-TAG). There are no niche optimizations in 0.1: an Option of a reference includes an explicit presence tag. Only the active alternative requires a valid value or owns resources; padding and inactive bytes are unspecified. These storage rules do not constitute a C enum or C union ABI. Target scalar and aggregate alignment remain implementation-defined (ID-TARGET).

Accepted struct attributes are @repr(C), @unsafe_send, @unsafe_sync, @derive(Json), and @json_deny_unknown; the latter requires @derive(Json). Derived JSON structs may rename fields with @json_name("name"). JSON derivation generates ordinary type- and borrow-checked methods for concrete structs; supported field types and encoding policies are specified in the [JSON reference](json.md). Thread contracts are described in the [thread reference](threads.md).

Enum declarations support only pub. Unknown, duplicate, or misplaced attributes/modifiers must be rejected. Attributes are not programmable macros.

16.3. Native ABI and runtime entry

ABI-NATIVE. Dodo-to-Dodo calling convention, linkage, and mangling are implementation-defined (ID-NATIVE). They must preserve the source types, evaluation, ownership, and return-source contracts. No cross-version or independent-compilation native ABI compatibility is promised. Stable foreign boundaries must use ABI-C with compatible storage.

ABI-ENTRY. Hosted executable emission requires a root, non-extern fn main() -> i32 or fn main() -> void (including omitted void return type). It supplies a C main wrapper that calls that function exactly once; a void result becomes zero, an i32 result becomes the C exit result. An external symbol named main conflicts with the wrapper and must be rejected. A library may omit main; check, object, assembly, bitcode, and IR emission must not demand a hosted entry signature or insert this wrapper. Platform startup, process exit-status encoding, and environment initialization are implementation-defined (ID-RUNTIME).

16.4. Assembly and barriers

Inline assembly, user alignment/section/export attributes, custom interrupt ABI, general target barriers, and custom panic-handler syntax are outside 0.1 and must not be silently accepted as those features. Unaligned data requires explicit unaligned operations; ordinary references with invalid alignment must not be created. Device/interrupt/DMA libraries must supply appropriate platform contracts.

17. Traps and arithmetic failure

TRAP. Safe integer overflow, division by zero, invalid shifts, and out-of-bounds indexing trap in every build profile. Numeric as conversions are explicit and checked where needed. Wrapping and truncation require named operations. Bounds checks may be removed only when proven unnecessary.

A freestanding target supplies entry and linker configuration. A required trap must not return to the failing operation or continue execution as if it succeeded. Abort does not unwind. The language guarantees neither real-time deadlines nor hardware correctness.

The trap instruction, diagnostic payload, and platform action are implementation-defined (ID-TRAP). The current implementation reports the failed check and source location before aborting on supported hosted targets, and uses llvm.trap by default on freestanding targets. LLVM may lower this intrinsic to C abort on targets without a trap instruction. Compiler options can select a non-returning C ABI board handler or trap-only behavior. The handler receives the failed check and source location and owns termination, with no fallback trap; returning from it is undefined behavior. This policy also applies to ordinary assertions; the hosted test runner retains its own reporting and trap behavior. No reset driver is supplied. See [runtime failure options](command-line.md). Compile-time constant failures receive diagnostics rather than an emitted runtime trap.

18. Diagnostics and implementation obligations

Borrow-conflict diagnostics should identify:

- The conflicting access.

- The loan's origin.

- The use that keeps the loan live.

- A safe repair direction, such as ending the loan sooner or separating disjoint data.

Diagnostics should not propose unsafe code as the default repair.

Before an implementation is described as memory-safe, its design and tests must cover reference aliasing, raw-pointer origins, mutation of borrowed fields, generic lifetime propagation, destruction, and interrupt/DMA interactions. This proposed borrowing model is not a soundness proof.

19. Worked examples

19.1. Structs, methods, and borrowing

This example demonstrates mutable and indexed foreach, method receivers, and a return borrow inferred from self, without allocation or explicit lifetime parameters.

```
package samples

pub struct Samples {
    values: [4]u16

    pub fn offset(&mut self, amount: u16) {
        for value in &mut self.values {
            *value += amount
        }
    }

    pub fn weighted_sum(&self) -> u32 {
        total := 0u32
        for i, &value in self.values {
            weight := i as u32 + 1
            total += weight * (value as u32)
        }
        return total
    }

    pub fn view(&self) -> &[u16] {
        return &self.values
    }
}

pub fn demo() -> u32 {
    samples := Samples{
        values: [1, 2, 3, 4],
    }

    view := samples.view()
    first := view[0]
    samples.offset(10)
    return samples.weighted_sum() + first as u32
}
```

view() returns a slice tied to self. Copying the scalar view[0] is the last use of the view, allowing the loan to end before offset(10). A later use of view would keep the shared loan live and require rejection of the mutation. Moving samples while its view remains live must also be rejected.

demo() evaluates to 131. In offset, value is &mut u16. In weighted_sum, value is &u16 and i is usize.

19.2. Result propagation and recovery

```
package hex

pub enum ParseError { Length, Digit }

fn nibble(ch: u8) -> u8!ParseError {
    match ch {
        b'0'..=b'9' => ok(ch - b'0'),
        b'a'..=b'f' => ok(ch - b'a' + 10),
        _ => err(ParseError.Digit),
    }
}

pub fn parse_byte(text: &[u8]) -> u8!ParseError {
    if text.len != 2 {
        return err(ParseError.Length)
    }
    let high = nibble(text[0])?
    let low = nibble(text[1])?
    ok((high << 4) | low)
}

pub fn parse_or(text: &[u8], fallback: u8) -> u8 {
    match parse_byte(text) {
        ok(value) => value,
        err(_) => fallback,
    }
}
```

u8!ParseError means Result<u8, ParseError>. parse_or(b"2a", 0) returns 42; parse_or(b"zz", 7) returns 7. Calls fail through declared result values or non-recoverable traps. ? marks propagation and possible early return; match marks local recovery.

19.3. An unsafe hardware boundary

```
package gpio
import "core/mmio"

pub struct Pin {
    set_address: usize
    clear_address: usize
    mask: u32

    // SAFETY: The caller supplies valid, aligned 32-bit SET/CLEAR registers,
    // a valid mask, and exclusive peripheral access for the handle's lifetime.
    // The peripheral must remain powered and configured while the handle exists.
    pub unsafe fn claim(set: usize, clear: usize, mask: u32) -> Pin {
        return Pin{set_address: set, clear_address: clear, mask}
    }

    pub fn high(&mut self) {
        // SAFETY: claim's contract guarantees a valid exclusive SET register.
        unsafe {
            mmio.write32(self.set_address, self.mask)
        }
    }

    pub fn low(&mut self) {
        // SAFETY: claim's contract guarantees a valid exclusive CLEAR register.
```

```
        unsafe {
            mmio.write32(self.clear_address, self.mask)
        }
    }
}

pub fn pulse(pin: &mut Pin, count: usize) {
    for _ in 0..count {
        pin.high()
        pin.low()
    }
}
```

Pin has private fields and moves rather than copies. high and low are safe only if the board layer establishes and preserves claim's contract. The registers are write-only SET/CLEAR aliases. The example does not authorize arbitrary read-modify-write operations or guarantee pulse timing.

Appendix A. Partial grammar

This EBNF-style summary is a reading aid for the normative rules above, not a complete parser specification. Lexing is defined in section 3 and expression precedence and sequencing in section 10. Historical accepted forms are described in their corresponding sections. A function without extern "C" requires a body; an extern function may have a body or end as a prototype.

```
program       = package-decl, { import-decl | declaration } ;
package-decl  = "package", identifier, newline ;
import-decl   = "import", string-literal, [ "as", identifier ], newline ;

declaration   = [ "pub" ],
                ( function-decl | struct-decl | enum-decl )
              | const-decl ;
const-decl    = "const", identifier, ":", type, "=", expression, newline ;

function-decl = [ "unsafe" ], [ "extern", string-literal ],
                "fn", identifier, [ type-params ], "(", [ parameters ], ")",
                [ "->", type ], [ borrow-source ], ( block | newline ) ;
parameters    = parameter, { ",", parameter } ;
parameter     = identifier, ":", type | "self" | "&", [ "mut" ], "self" ;
borrow-source = "from", "(", ( "static" | identifier-list ), ")" ;

struct-decl   = "struct", identifier, [ type-params ], "{",
                { field-decl | function-decl }, "}" ;
field-decl    = [ "pub" ], identifier, ":", type, newline ;
enum-decl     = "enum", identifier, [ type-params ], "{",
                enum-variant, { ",", enum-variant }, "}" ;

type          = primitive-type
              | identifier, [ type-args ]
              | "[", constant-expression, "]", type
              | "&", [ "mut" ], type
              | "&", [ "mut" ], "[", type, "]"
              | "*const", type | "*mut", type
              | type, "!", type ;

block         = "{", { statement }, "}" ;
statement     = local-decl | expression-stmt | return-stmt
              | if-stmt | for-stmt | match-stmt
              | break-stmt | continue-stmt | unsafe-block ;
local-decl    = "let", pattern, [ ":", type ], "=", expression,
```

```
                [ "else", block ], newline
              | identifier, ":=", expression, newline
              | identifier, ":", type, [ "=", expression ], newline ;
return-stmt   = "return", [ expression ], newline ;
if-stmt       = "if", ( expression | "let", pattern, "=", expression ), block,
                { "else", "if", expression, block },
                [ "else", block ] ;
for-stmt      = "for", block
              | "for", expression, block
              | "for", for-init, ";", expression, ";",
                expression, block
              | "for", identifier, "in", expression, [ "..", expression ], block
              | "for", [ identifier, "," ], [ "&" ], identifier,
                "in", expression, block ;
match-stmt    = "match", expression, "{", { match-arm }, "}" ;
match-arm     = pattern, [ "if", expression ], "=>",
                ( expression | block ), [ "," ] ;
pattern       = pattern-atom, { "|", pattern-atom } ;
pattern-atom  = "_" | identifier | boolean-literal | integer-literal
              | integer-literal, ( ".." | "..=" ), integer-literal
              | qualified-name, "(", [ pattern, { ",", pattern } ], ")"
              | qualified-name, "{", [ pattern-field, { ",", pattern-field } ], "}"
              | "(", pattern, ")" ;
pattern-field = identifier, [ ":", pattern ] | ".." ;
unsafe-block  = "unsafe", block ;
array-list    = "[", [ expression, { ",", expression }, [ "," ] ], "]" ;
array-literal = array-list
              | "[", expression, ";", constant-expression, "]" ;
typed-array   = "(", array-list, ":", array-type, ")" ;
struct-field  = identifier, [ ":", expression ] ;
subslice      = expression, "[", [ expression ], "..", [ expression ], "]" ;
match-expr    = "match", expression, "{", { value-arm }, "}" ;
value-arm     = pattern, [ "if", expression ], "=>",
                ( expression | value-block ), [ "," ] ;
value-block   = "{", { statement }, expression, "}" ;
```

Appendix B. Conformance checklist

A compiler claiming Dodo 0.1 conformance must, at minimum:

- Accept the specified declarations, functions, methods, control flow, and types.
- Default omitted return types to void; accept final-expression and explicit returns.
- Distinguish immutable runtime bindings, mutable locals, and compile-time constants.
- Share recursive patterns across matching and conditional bindings without weakening Result handling.
- Infer local generic arguments and array elements where the context suffices.
- Support explicit copy patterns only for shared copyable collection elements.
- Preserve explicit types when formatting historical syntax into canonical forms.
- Support leading-dot continuation with semicolons as expression boundaries.
- Resolve constant dependencies and reject cycles and invalid array lengths.
- Preserve evaluation order, ownership, and cleanup in value blocks and ranges.
- Enforce definite initialization and invalidation after moves.

- Enforce shared-versus-exclusive borrowing for overlapping memory.
- Check borrowed-return contracts and reject references escaping destroyed locals.
- Destroy owned values deterministically on normal exits, including ?, break, and continue.
- Reject discarded Result values.
- Restrict unchecked operations to the specified unsafe boundary.
- Trap on safe overflow, division by zero, invalid shifts, and invalid indexing in every build profile.
- Preserve volatile-access semantics.
- Avoid a mandatory garbage collector, heap, scheduler, or exception runtime when the program does not request one.

These requirements alone do not constitute a proof of soundness.

B.1. Rule-to-test index

A means acceptance, R required rejection, E execution at both -O0 and -O3, and T acceptance plus inspection of emitted target IR. T establishes lowering properties, not execution on that target. Paths below are relative to the repository; names after :: are Rust test functions, usable as Cargo test filters. Suite abbreviations refer to these files:

- spec: [tests/specification.rs](https://github.com/Jotrorox/dodo/blob/main/tests/specification.rs)
- compiler: [tests/compiler.rs](https://github.com/Jotrorox/dodo/blob/main/tests/compiler.rs)
- regressions: [tests/regressions.rs](https://github.com/Jotrorox/dodo/blob/main/tests/regressions.rs)
- storage: [tests/owned_storage.rs](https://github.com/Jotrorox/dodo/blob/main/tests/owned_storage.rs)
- core: [tests/core_intrinsics.rs](https://github.com/Jotrorox/dodo/blob/main/tests/core_intrinsics.rs)
- packages: [tests/stdlib_packages.rs](https://github.com/Jotrorox/dodo/blob/main/tests/stdlib_packages.rs)
- lexer, parser, sema, package: the tests modules in src/lexer.rs, src/parser.rs, src/sema.rs, and src/package.rs respectively.

Each row is a conformance obligation, not merely a record of what happens to compile. Undefined executions (dangling access, invalid tags, mismatched foreign allocators, data races, or invalid pointer/owner correspondence) are deliberately not executed as tests expecting rejection or a trap. Their adjacent valid execution cases and enforced static boundaries are identified separately.

Rule or section	Acceptance, rejection, or execution evidence

PKG-UNIT	E/R compiler::cli_projects_default_to_main_and_compile_keeps_an_executable, cli_project_folders_and_libraries_are_ordinary_local_imports, cli_missing_project_entry_does_not_search_parents_or_other_files; E spec::directory_package_scope_aliases_and_constant_initialization, canonical_package_identity_and_symlink_enumeration (Unix); R spec::package_paths_cycles_ambiguity_and_unit_mismatch_are_rejected; A package::root_directory_collects_only_immediate_dodo_files.
PKG-RESOLVE	R spec::package_paths_cycles_ambiguity_and_unit_mismatch_are_rejected; A packages::local_packages_can_import_stdlib_and_stdlib_dependencies_are_deduplicated.
PKG-NAMES	E/R spec::directory_package_scope_aliases_and_constant_initialization; A/R packages::aliases_distinguish_same_named_packages_and_transitive_dependencies; R/A package::transitive_packages_require_a_direct_import.
PKG-CORE	A/R packages::bundled_imports_cannot_be_shadowed_by_local_files_or_editor_overlays, unknown_and_malformed_stdlib_imports_are_rejected_without_local_fallback, local_packages_cannot_conflict_with_bundled_or_intrinsic_aliases, transitive_imports_do_not_grant_source_or_intrinsic_package_visibility.
PKG-VIS	E compiler::directory_packages_import_public_types_fields_and_methods; R compiler::imports_reject_private_functions_fields_and_field_construction; R sema::public_api_cannot_expose_private_type.
PKG-INIT	E spec::static_storage_has_no_implicit_initialization_or_destruction_hooks; E/R spec::directory_package_scope_aliases_and_constant_initialization; E compiler::constants_compose_and_define_array_types; R compiler::constant_cycles_width_and_expansion_limits_are_diagnosed.
LEX-TEXT, LEX-LITERAL, section 3.1	E/R spec::lexical_boundaries_and_literal_encodings; A lexer::numeric_literals_and_longest_operators, utf8_and_byte_escapes, comments_preserve_newlines_and_byte_spans; R lexer::malformed_source_returns_diagnostics; A/R/E tests/newline_continuation.rs.

EXPR-PREC	E spec::evaluation_order_and_precedence; A parser::precedence_casts_and_postfix; R compiler::new_forms_reject_invalid_types_moves_lifetimes_and_unhandled_errors.
EXPR-ORDER	E spec::evaluation_order_and_precedence, compiler::floats_casts_and_short_circuit_preserve_side_effects, inferred_arrays_and_repetition_evaluate_once_even_when_empty, range_bounds_are_captured_once_and_iteration_bindings_are_fresh, slice_source_and_bounds_evaluate_once_in_order.
EXPR-ORDER cleanup	E regressions::propagation_drops_previously_evaluated_call_arguments, propagation_drops_moved_call_arguments_exactly_once, propagation_drops_initialized_struct_fields, propagation_drops_initialized_array_elements, propagation_drops_initialized_enum_payloads, returning_a_void_call_preserves_effects_and_cleanup; E compiler::value_expression_exits_cleanup_on_return_break_and_continue.
EXPR-ASSIGN	E spec::assignment_captures_destination_and_old_value_before_rhs; E core::replacement_failure_preserves_original_and_drops_it_once.
REP-SCALAR, NUM-FLOAT	E spec::scalar_representations_and_float_comparisons; T spec::target_profiles_publish_width_endianness_and_native_symbols.
REP-TAG, REP-VIEW, REP-LAYOUT	E spec::aggregate_layout_tags_and_string_byte_lengths, shared_payload_values_survive_moves_borrows_and_propagation, shared_payload_cleanup_drops_only_the_active_alternative; T spec::shared_payload_layout_on_native_and_embedded_targets; E debugging::gdb_reads_shared_enum_and_result_payloads; R spec::unsafe_pointer_conversions_and_unsupported_abi_are_rejected; R core::field_offsets_enforce_literal_direct_fields_and_package_privacy.
NUM-CAST, NUM-ARITH, TRAP	E spec::scalar_representations_and_float_comparisons, checked_float_cast_boundaries_trap; E compiler::arithmetic_and_index_traps_survive_optimization, invalid_subslice_bounds_trap_at_all_optimization_levels; R sema::constant_arithmetic_checked_before_codegen; E/R/T core::wrapping_arithmetic_boundaries_inference_and_evaluation_order, wrapping_arithmetic_rejects_invalid_arguments, wrapping_arithmetic_emits_plain_operations_at_each_target_width.

GEN-INSTANCE, section 4.7	E compiler::generic_arguments_infer_from_values_parameters_and_return_context, generic_struct_methods_keep_borrowed_returns; R compiler::new_forms_reject_invalid_types_moves_lifetimes_and_unhandled_errors.
OWN-SUBOBJECT, sections 7–9	R sema::destructuring_preserves_borrows_moves_and_custom_drop; E compiler::cleanup_reverses_order_and_does_not_double_drop_moves, cleanup_runs_on_overwrite_explicit_drop_break_and_continue, custom_destructor_runs_before_nested_fields, nested_patterns_drop_ignored_values_and_guard_failures_once.
PTR-VALID, PTR-OFFSET	E valid ranges/reinterpretation spec::pointer_integer_round_trip_offsets_and_byte_access, scalar_representations_and_float_comparisons; E foreign allocated storage storage::simultaneous_boxes_destruction_failure_alignment_and_zst. Invalid runtime provenance is an unsafe precondition violation, not a required rejection.
PTR-CAST	E spec::pointer_integer_round_trip_offsets_and_byte_access; R spec::unsafe_pointer_conversions_and_unsupported_abi_are_rejected, sema::raw_pointer_borrow_cannot_invent_a_checked_lifetime.
PTR-ALIAS, PTR-VIEW	E spec::pointer_integer_round_trip_offsets_and_byte_access, owner_view_arguments_keep_their_effects_and_order; R storage::owned_views_are_unsafe_to_construct_and_require_matching_access, views_cannot_escape_owner_or_erase_underlying_dependencies, live_views_prevent_replacement_removal_and_destruction, opaque_views_reject_nested_reference_and_result_payloads; E/R storage::shared_dependencies_do_not_become_exclusive_when_mutating_owned_fields. These reject checked conflicts; they do not dynamically validate raw accesses.
PTR-BYTES, CORE-MEM	E spec::pointer_integer_round_trip_offsets_and_byte_access, core::core_storage_pointers_and_owned_exchange, opaque_storage_moves_without_dropping_contents; R core::pointer_and_storage_errors_are_diagnostics.
CORE-MMIO, sections 15.2–15.3	T compiler::gpio_emits_volatile_accesses_without_host_execution; R sema::mmio_checked; board address/width legality remains an unsafe precondition.

ABI-C	E spec::c_scalar_results_and_struct_pointer_layout_interoperate; R spec::incompatible_lowered_foreign_declarations_fail_before_linking; E regressions::foreign_narrow_integer_arguments_follow_the_c_abi, compiler::hello_uses_native_c_abi, imported_foreign_declarations_keep_their_c_symbol_name, repeated_foreign_declarations_across_packages_share_a_c_symbol; E/R spec::hosted_entry_signatures_and_c_definitions; R spec::unsafe_pointer_conversions_and_unsupported_abi_are_rejected.
ABI-NATIVE, ABI-ENTRY	T spec::target_profiles_publish_width_endianness_and_native_symbols; A/R/E spec::hosted_entry_signatures_and_c_definitions; E compiler::cli_run_preserves_program_exit_status; T compiler::emits_llvm_ir_bitcode_assembly_and_object_files.
Bindings, returns, patterns, sections 5–6 and 11–12	E compiler::immutable_runtime_bindings_preserve_mutable_references_and_function_tails, recursive_patterns_ranges_guards_and_conditional_bindings_execute, alternative_patterns_retry_guards_in_order; R compiler::new_binding_and_pattern_forms_reject_invalid_programs, rejects_unhandled_results_and_incompatible_propagation; A/R/E tests/reference_iteration.rs.
Borrow contracts and unsafe boundary, sections 8, 13	E compiler::borrowed_return_contracts_and_static_storage, references_reborrow_and_disjoint_fields; R compiler::rejects_conflicting_borrows_and_moves, rejects_escaping_or_ambiguous_borrows, unsafe_function_body_still_needs_an_explicit_block.
Diagnostics, section 18	A/R tests/diagnostics.rs, tests/lsp.rs; R package::diagnostic_labels_resolve_each_file_independently.

Run the complete repository evidence with cargo test --locked --all-targets. The focused new contracts run with cargo test --locked --test specification. Native execution evidence is for the host tested; target emission alone does not establish another target's ABI or hardware behavior.

Appendix C. Implementation-defined behavior and excluded features

C.1. Required implementation profile

An implementation must publish choices for the IDs below, including the compiler version and target. Changing an implementation-defined choice must not silently change the source-level rules above. The following is the compiler 0.1.4 profile; tests constrain the documented choice, not every possible conforming choice.

Choice	Compiler 0.1.4 definition and evidence

ID-TARGET: widths, endianness, alignment, object format	Uses the selected LLVM 23 target triple and its data layout. Default is the compiler host triple. x86_64-unknown-linux-gnu has 64-bit pointers and little-endian storage; i686-unknown-linux-gnu has 32-bit pointers and little-endian storage; powerpc64-unknown-linux-gnu has 64-bit pointers and big-endian storage. Scalar/aggregate alignment is the target data layout's ABI alignment, observable with core layout queries; emitted IR must include the triple/data layout. Evidence: spec::target_profiles_publish_width_endianness_and_native_symbols (T), aggregate_layout_tags_and_string_byte_lengths (E).
ID-FLOAT: floating environment and constant precision	Runtime uses LLVM non-fast floating operations with default round-to-nearest, ties-to-even. Native tests assume gradual underflow and masked floating exceptions; no source facility changes rounding/exception modes. Constant scalar evaluation computes floating operations in host binary64 and rounds to the expression's type at each node. Integer-to-float constants round directly to the destination representation using round-to-nearest, ties-to-even, matching runtime conversion; an explicit cast through f64 still performs both conversions. Decimal literal conversion passes through binary64. Only narrowing f64 to f32 requires a finite, in-range source; identity and widening floating casts preserve NaN, infinities, and signed zero in constants and at runtime. NaN payload/sign is unspecified. Evidence: spec::scalar_representations_and_float_comparisons, checked_float_cast_boundaries_trap, constant_integer_to_float_matches_runtime, constant_float_identity_and_widening_match_runtime (E); numeric_conversions::constant_float_narrowing_still_rejects_non_finite_and_out_of_range_values (R).
ID-PTR: address representation	The integral default-address-space profile uses LLVM data pointers and integer casts, with zero as null; pointer offsets use element-scaled address computation. No hidden runtime provenance table is maintained. Non-default address spaces and capability/non-integral pointer interfaces are outside this profile; LLVM emission alone does not establish their conformance. Evidence: spec::pointer_integer_round_trip_offsets_and_byte_access (E), unsafe_pointer_conversions_and_unsupported_abi_are_rejected (R).

ID-C: target C argument/result passing	Uses LLVM C calling convention. On x86 SysV, i8/i16 use sign extension and u8/u16/bool use zero extension on declarations and calls; Windows uses its target convention without those SysV attributes. Other scalar and pointer passing follows the selected target. Duplicate extern declarations are compared by lowered LLVM function type, which does not distinguish signed/unsigned integers of the same width; the author must keep those contracts consistent. Evidence: spec::c_scalar_results_and_struct_pointer_layout_interoperate (E), incompatible_lowered_foreign_declarations_fail_before_linking (R), regressions::foreign_narrow_integer_arguments_follow_the_c_abi (E), compiler::repeated_foreign_declarations_across_packages_share_a_c_symbol (E).
ID-NATIVE: Dodo ABI and symbols	Parameters/results use the scalar/aggregate LLVM types in sections 4.8 and 16.2 with LLVM's default calling convention and no C aggregate classification. Internal parameters/results larger than 1,024 target ABI bytes pass indirectly through caller-owned storage; exported and C signatures retain direct lowering. Symbols are dodo.<root-package>.<qualified-name>. Root declarations have no package prefix in qualified-name; imported packages use their name when unique and generated __dodo_package_... prefixes when necessary. Root-package public non-generic functions and methods, root main, and all C ABI definitions have external linkage; imported Dodo functions and generic instances have internal linkage. Unreachable internal functions are eliminated at every optimization level, following lowered references including implicit drops and callback addresses. ELF and COFF functions use separate code sections for linker garbage collection. Mangling of generic instances and collision prefixes is compiler-internal and may depend on the loaded graph. Link native objects only with a matching compiler/target configuration. Evidence: spec::target_profiles_publish_width_endianness_and_native_symbols (T), compiler::generic_struct_methods_keep_borrowed_returns (E).
ID-FS: filesystem identity	Uses the host filesystem's path canonicalization and case sensitivity; directory entries are sorted using canonical host paths. Symlink entries are excluded from directory source enumeration, but an explicit input/import path may resolve through a symlink. No registry, network resolver, or lockfile participates. Evidence: spec::canonical_package_identity_and_symlink_enumeration (E, Unix), package::root_directory_collects_only_immediate_dodo_files (A), spec::package_paths_cycles_ambiguity_and_unit_mismatch_are_rejected (R).

ID-LIB: bundled platform selection	Imports are embedded in the compiler. std/{platform,fs,process,env,thread,sync,net,tls,web}/native selects linux for x86_64 Linux GNU and windows for x86_64 Windows GNU/MSVC. An explicit mismatching adapter or unsupported target is rejected. Portable packages need no hosted adapter. Evidence: tests/platform_library.rs and packages::portable_package_dependencies_are_independent_and_reserved (A/R).
ID-RUNTIME: startup and linking	Hosted builds use a native C linker driver. Importing std/env initializes its argument runtime from the C main arguments before source main; other library facilities initialize through explicit calls. Raw object/IR emission supplies no startup. Freestanding startup, linker scripts, system libraries, and OS exit-status encoding are supplied by the target/toolchain. Evidence: spec::hosted_entry_signatures_and_c_definitions (A/R/E), tests/env_library.rs (E), compiler::emits_llvm_ir_bitcode_assembly_and_object_files (T).
ID-TRAP: failure mechanism	--panic auto writes the check kind, file, line, and column to stderr and calls C abort on supported hosted targets; other targets use llvm.trap, which may itself lower to C abort. --panic hosted, --panic trap, and --panic-hook SYMBOL select behavior explicitly for checked operations and ordinary assertions. The non-returning C ABI hook receives static NUL-terminated check/file strings and u32 line/column and owns termination without a fallback trap; returning is undefined behavior. The hosted test runner uses its own reporting and trap behavior. No mode unwinds, runs destructors, or supplies a reset driver. Instruction, signal/exception, and exit encoding are target-specific. Evidence: tests/debugging.rs, compiler::arithmetic_and_index_traps_survive_optimization, spec::checked_float_cast_boundaries_trap (E/T).
ID-HW: device access and atomics	MMIO widths are restricted as in CORE-MMIO and use volatile loads/stores; board address legality is an unsafe precondition. Native integer atomic capabilities/orderings are target-restricted; unsupported operations are diagnosed, with no implicit libatomic fallback. Evidence: compiler::gpio_emits_volatile_accesses_without_host_execution (T), tests/thread_library.rs::atomic_ordering_and_target_capabilities_are_checked (R/T).

ID-LIMIT: resource limits	Parser nesting: 64 levels. Constant dependency depth: 128; expansion work: 200,000 nodes. Array count must fit target usize and u32; nonzero repeated constants are limited to 1,000,000 elements. Generic instantiation: 4,096 specializations and 64 nested specializations (including generated methods). Pattern expansion: 4096 alternatives; exhaustiveness work has a 131,072-unit budget (charged for matrix rows and type columns); exhaustion must diagnose, not assume coverage. Evidence: compiler::generic_specialization_breadth_and_depth_have_separate_limits (R), compiler::constant_cycles_width_and_expansion_limits_are_diagnosed (R), sema::excessive_pattern_expansion_reports_a_diagnostic (R), compiler::new_forms_reject_invalid_types_moves_lifetimes_and_unhandled_errors (R).
ID-DIAG: diagnostic presentation	Errors identify a source location where available. Text wording, ordering, number of follow-on errors, and CLI/LSP presentation are not a stable machine-readable language ABI. Editor transport is separately documented. Evidence: tests/diagnostics.rs, tests/lsp.rs, package::diagnostic_labels_resolve_each_file_independently (A/R).

C.2. Features excluded from this edition

Trait constraints, specialization, user lifetime/variance syntax, standalone partial field moves, a package registry, package initializers, re-exports, variadic/C aggregate-by-value calls, inline assembly, user section/alignment/export attributes, interrupt ABI declarations, and panic-handler registration are not part of 0.1. The compiler must reject syntax presented as these features rather than assign it an undocumented meaning. Checked mutable slice splitting is available through core/slice.split_at_mut and its compiler-recognized SplitMut<T> result. General user-defined disjointness proofs, DMA abstractions, and interrupt-safe allocators require future library/compiler work. The named rules above replace the former open questions about fundamental execution, representation, provenance, ABI, and packages; they do not assert a formal proof of memory safety or complete platform support.

Appendix D. Informative references

The supplied design cites these influences. They are informative, not normative Dodo rules, and this edition does not incorporate their contents by reference.

- Go specification, “For statements”: https://go.dev/ref/spec

- Rust Reference, “Lifetime elision”: https://doc.rust-lang.org/reference/lifetime-elision.html

- Rust core::result: https://doc.rust-lang.org/core/result/

- Rust Reference, “The unsafe keyword”: https://doc.rust-lang.org/reference/unsafe-keyword.html

- Rust core::ptr::read_volatile: https://doc.rust-lang.org/core/ptr/fn.read_volatile.html

End of Dodo Language Specification 0.1.