Tamago Programming Language Official Specification v1.0

Overview

Tamago is a deterministic, modular, and OS/freestanding-compatible programming language designed for robust system-level and embedded development. Syntax and semantics are fully explicit, enabling unambiguous memory management and symbol control. Inline assembly is strictly restricted to marked segments. Identical program and build environments produce bit-identical outputs, regardless of platform (OS-agnostic, embedded, or TamaOS-native).

---

1. Goals

1.1 Business Goals

- **Deterministic ecosystem:** Toolchain implementations must produce byte-identical results with the same inputs and explicit build environments. No invisible transformations permitted.
- **Cross-platform efficiency**: Core language constructs must be portable if, and only if, platform or extension annotations permit. Implicit API broadenings are strictly prohibited.
- **Library partitioning:** Compiler and linker are required to enforce clear boundaries between core and user APIs, with privilege leakage producing diagnostic errors.
- **OS/freestanding adaptability**: Built-in targeting of boot, microkernel, and embedded ISAs as first-class use cases. General-purpose OS support never presupposed.
- **Security and safety:** Granular privilege, flag, and attribute mechanisms must be enforced; silent privilege escalation or memory ambiguities are forbidden.

1.2 User Goals

- **Deterministic builds:** Every artifact must be unaffected by host or ambient environment.
- **Explicit type/layout rules:** All memory, storage, addresses, and alignments must be explicit and uniformly interpreted by conforming implementations.
- **Namespaced modularity:** Hierarchical modules and safe symbol containment are strictly required, supporting definable decoupling.
- **Privileged features, regulated:** Inline assembly and hardware access are available only with explicit, compiler-verified flagging.

- **Scoped library exposure:** Namespace and library imports outside valid/compatible contexts must be diagnosed as violations.

1.3 Non-Goals

- No dynamic reflection, memory “safety by default,” or runtime metaprogramming
- No JIT, plugins, or runtime module loading
- No type erasure or auto inference (all types, conversions explicit)
- No compiler optimization unless explicitly directed by the user

---

2. User Stories

- **System developer:** Implements a bare-metal loader—direct register access, with OS presence never assumed.
- **Embedded engineer:** Depends on transparent, deterministic binary/SYMTAB structure across ARM/RISC-V/custom ISAs.
- **User-mode developer:** Expects compile-time enforcement against unsafe (privileged) APIs in user-space modules.
- **Toolchain maintainer:** Needs total ABI visibility and diagnostics with actionable causes and cross-references.
- **Contributor:** Onboards via comprehensive, actionable diagnostics, including improvement hints and fixes.

---

3. Functional Requirements

3.1 Lexical Analysis & Syntax

3.1.1 Lexical Rules

- **Tokenization:** All keywords listed in Annex B are reserved and blocked as identifiers. Use as user symbol triggers `E0001`
- **Identifiers:** ASCII letters, underscores, non-leading digits only. (No Unicode or escapes.)
- **Literals:**
  - Integers must use their base explicitly (`0x`, `0b`, `0d`), and width suffix for non-standard types (e.g., `5u16`).
  - Octal literals (e.g., `0o755`) cause `E0012`.
  - Unicode escapes (`\uXXXX`) in string literals only; not in identifiers or character literals.
- **Whitespace:** Collapsed except within strings/attributes. Newlines are always statement/block terminators.
- **Line endings:** All non-`\n` line endings emit `W0009`; compilers normalize for tokenization.
- **Full EBNF formal grammar: see §3.2 and Annex B.**

3.1.2 Forbidden Constructs (**Normative**)

- Reserved keyword as name or symbol – Error `E0001`

- Octal literals, implicit width – Error `E0012`
- Dangling/illegal line joins – Error `E0220`
- Declaration outside namespace – Error `E2001`

3.1.3 Compatibility Tiers

- **Universal:** Core lexical and syntactical rules are invariant and must be implemented identically.
- **Extension:** Platform-specific input rules (e.g., Unicode normalization, exotic whitespace) allowed if fully orthogonal and explicitly documented.

3.2 Formal Grammar

3.2.1 Complete ISO-Style EBNF Syntax for Tamago

Start symbol: program

ISO/IEC EBNF:

```
program ::= { namespace_decl | import_stmt | include_stmt }
```

```
namespace_decl ::= attribute_seq? "namespace" identifier "{"
{ decl_or_stmt } "}"
```

```
attribute_seq ::= { attribute }
```

```
attribute ::= "#[" attr_target "::" identifier ( "::" identifier )*
"]"
```

```
attr_target ::= "attribute" | "flag"
```

```
import_stmt ::= "use"namespace_path ";"
```

```
namespace_path ::= identifier { "." identifier }
```

```
include_stmt ::= "include" string_literal ";"
```

```
decl_or_stmt ::= decl | statement
```

```
decl ::= var_decl | const_decl | fn_decl | proc_decl | struct_decl |
enum_decl
```

```
var_decl ::= attribute_seq? "var" identifier ":" type [ "=" expr ]
";"
```

```
const_decl ::= attribute_seq? "const" identifier ":" type [ "="
expr ] ";"
```

```
fn_decl ::= attribute_seq? "fn" identifier "(" param_list? ")"
fn_ret_ty block
```

```
proc_decl ::= attribute_seq? "proc" identifier "(" param_list? ")"
proc_ret_ty block
```

```
param_list ::= param { "," param }
```

```
param ::= type identifier ( " #[attribute::variadic]" )?

fn_ret_ty ::= "->" type | "->" "void" proc_ret_ty ::= "->" type | "-
>" "void"

struct_decl ::= "struct" identifier "{" struct_fields "}"

struct_fields ::= [ struct_field { "," struct_field } ]

struct_field ::= type identifier

enum_decl ::= "enum" identifier "{" enum_members "}"

enum_members ::= [ enum_member { "," enum_member } ]

enum_member ::= identifier [ "=" expr ]

statement ::= expr_stmt | assignment | ctrl_stmt | block | asm_block

block ::= "{" { decl_or_stmt } "}"

expr_stmt ::= expr ";"

assignment ::= lhs assign_op expr ";"

lhs ::= identifier | array_access | struct_access | pointer_deref

assign_op ::= "=" | "+=" | "-=" | "*=" | "/="

ctrl_stmt ::= if_stmt | while_stmt | for_stmt | break_stmt |
continue_stmt | return_stmt

if_stmt ::= "if" "(" expr ")" block [ "else" block ]

while_stmt ::= "while" "(" expr ")" block

for_stmt ::= "for" "(" [expr_stmt] [expr] ";" [expr] ")" block

break_stmt ::= "break" ";"

continue_stmt ::= "continue" ";"

return_stmt ::= "return" [expr] ";"

mnemonic ::= /* See Annex B, Assembly Mnemonics */

operand ::= identifier | int_literal | register register ::= "R0"
| ... | "R15" | "SR0" | ... | "SR3" | "BP" | "SP" | "PC" | "FLAGS" |
"SYS"

expr ::= logic_or_expr
```

// --- Expression Grammar with Precedence, Associativity (Explicit Table Below) ---

```
logic_or_expr ::= logic_and_expr { "||" logic_and_expr } /* left /

logic_and_expr ::= equality_expr { "&&" equality_expr } / left /

equality_expr ::= rel_expr { ( "==" | "!=" ) rel_expr } / left /
```

```
rel_expr ::= bitwise_or_expr { ( "<" | "<=" | ">" | ">=" )
bitwise_or_expr } / left /

bitwise_or_expr ::= bitwise_xor_expr { "|" bitwise_xor_expr } / left
/

bitwise_xor_expr ::= bitwise_and_expr { "^" bitwise_and_expr } /
left /

bitwise_and_expr ::= shift_expr { "&" shift_expr } / left /

shift_expr ::= additive_expr { ( "<<" | ">>" ) additive_expr } /
left /

additive_expr ::= multiplicative_expr { ( "+" | "-" )
multiplicative_expr } / left /

multiplicative_expr ::= unary_expr { ( "" | "/" | "%" ) unary_expr }
/* left /

unary_expr ::= ( "+" | "-" | "!" | "" | "~" ) unary_expr /* right /
| cast_expr | primary

cast_expr ::= "cast" "<" type ">" "(" expr ")" / right */

primary ::= literal | identifier | fn_call | array_access |
struct_access | pointer_deref | "(" expr ")"

// --- Additional Expression Forms, Arrays, Pointers, Structs, Type
names ---

fn_call ::= identifier "(" arg_list? ")"

arg_list ::= expr { "," expr }

array_access ::= identifier "[" expr "]" struct_access ::=
identifier "." identifier

pointer_deref ::= "*" expr

// --- Type System, Array/Pointer Syntax, Casts ---

type ::= base_type | type "[" int_literal "]" /* array type / | ""
type /* pointer type / | identifier / struct/enum/user type */

base_type ::= "int8" | "int16" | "int32" | "int64" | "uint8" |
"uint16" | "uint32" | "uint64" | "float32" | "float64" | "char" |
"bool" | "string" | "pointer"

literal ::= int_literal | float_literal | string_literal |

char_literal | bool_literal | array_init | struct_init

int_literal ::= ( "0d" | "0x" | "0b" ) digits [ int_width ]

int_width ::= "u8" | "u16" | "u32" | "u64" | "i8" | "i16" | "i32" |
"i64"

float_literal ::= digits "." digits [ float_width ]
```

`float_width ::= "f32" | "f64" string_literal ::= """ { char | escape_sequence } """`

`char_literal ::= "'" ( char | escape_sequence ) "'"`

`bool_literal ::= "true" | "false"`

`array_init ::= "[" [ expr { "," expr } ] "]"`

`struct_init ::= identifier "{" [ struct_field_init { "," struct_field_init } ] "}"`

`struct_field_init ::= identifier ":" expr`

`digits ::= digit { digit }`

`digit ::= "0" .. "9"`

`escape_sequence ::= "\n" | "\t" | "\\" | "\"" | "\'" | "\x" hex_digit hex_digit | "\u" hex_digit hex_digit hex_digit hex_digit`

`identifier ::= identifier_start identifier_body identifier_start ::= "" | ascii_letter`

`identifier_body ::= { "" | ascii_letter | digit } ascii_letter ::= "a" .. "z" | "A" .. "Z"`

`hex_digit ::= "0" .. "9" | "A" .. "F" | "a" .. "f"`

/* Restrictions: identifier may begin with single '`_`', not '`__`' (see §11.1.2, error `E0260`) */

/* Note (v1.0): In `#[flag::freestanding]` contexts, fn/proc block bodies may alternatively be tamassembly_body:

`tamassembly_body ::= { (asm_line | label_decl | comment_line) }`

`asm_line ::= mnemonic { operand } ";"`

`label_decl ::= identifier ":"`

`comment_line ::= ";" { any text }`

This is a semantic restriction (see §9). Mnemonics, operands, and registers are fixed as defined in Annex B. */

(v1.0 REVIEW: This grammar is ISO-style, normative, and covers the complete Tamago language as implemented and tested. Operator associativity/precedence matches §11.1.1.)

Attribute Placement Clarification:*Attributes (e.g. `#[attribute::cstring]`, `#[flag::user]`) always precede the declaration keyword or parameter type/name.*

Example:`#[attribute::cstring] const s: string = "abc\x00";fn debug(msg: string, #[attribute::variadic] rest: int32) -> void { ... }`

Attributes must be outside the parameter list if attached to the declaration.Attribute usage is EBNF-enforced and statically diagnosed (see error E1281*, EBNF productions above).*

3.2.2 Operator Precedence/Associativity Table

Level	Operators	Assoc.	Description
1	cast(expr)	right	Type cast (explicit)
2	*, -, !, ~, + (unary)	right	Unary
3	*, /, %	left	Multiplicative
4	+, -	left	Additive
5	<<, >>	left	Bit shifts
6	&	left	Bitwise AND
7	^	left	Bitwise XOR
8	`	left	`
9	<, <=, >, >=	left	Comparison
10	==, !=	left	Equality
11	&&	left	Logical AND
12	`	left	`
13	=, +=, -=, *=, /=, etc.	right	Assignment
(Table is explicit; all operators are non-ambiguous; cross-list in Annex B.)			

3.3 Diagnostics & Error Handling

3.3.1 Diagnostic Obligations (see Annex G)

- **Each diagnostic must include:**
  - Unique error code, error class label, filename, line, column
  - Human-readable explanation, specific spec/annex section reference
  - Actionable remediation/fix hints
  - Colorized/annotated output (required for GUI/modern toolchains)

3.3.2 Examples (Verbatim Structure)

```
error[E0001]: reserved keyword used as identifier (§3.1.2, Annex
B)--> sys.svc.tmg:8:11|8 | namespace MOV {| ^^^ reserved
keywordsuggestion: replace with non-keyword name (see Annex B)

error[E8001]: declaration outside namespace (§3.2.2)-->
boot/start.tmg:1:11 | int32 valid = 0;| ^^^^^^^^^^^^^^suggestion:
syntactically nest this in a namespace, per §3.2.1
```

3.3.3 Error Classes (Full taxonomy, Annex G)

- **Error**: Hard failure; processing stops, must fix.
- **Warning**: Compilation continues; should fix.
- **Hint**: Advisory, non-blocking.
- **Deprecation**: Forgetting will become error in future versions.

3.3.4 Undefined Behavior Diagnostics

- All potential undefined behavior situations (null deref, out-of-bounds, use-after-free, etc.) must be diagnosed at compile time where statically possible, issuing warning (`W100x`) and referencing the normative rules per §5, §11, and Annex E.
- Examples:
    - Null pointer dereference at compile time: `warning[W1001]: null pointer dereference (see §5.3, §11)`
    - Out-of-bounds static array access: `warning[W1002]: out-of-bounds access (see §5.3, §11)`
- Tools should highlight these regions; colorization is recommended (not required).

3.4 Cross-Platform, Extension, Standard Library (Normative)

3.4.1 Universal Core

- All base syntax, typing, diagnostics, program structure, error behaviors, and standard library primitives (see §10) are invariant.

3.4.2 Platform-Specific/Extension Syntax

- See extension attributes, manifests; new forms must always reference allowed ABI.

3.4.3 Standard Library Coverage

Required minimal APIs (See §10 for details):

- **Memory:** `memcpy`, `memset`, `memcmp`
- **Environment:** `env_query` (for freestanding, must fail gracefully if not present)

- **String APIs:** explicit conversions, bounded copy
- All APIs must have explicit argument types, error/overflow handling, and must never perform hidden conversions.

3.4.4 Vendor Extension

- Each vendor/extension must provide a manifest (see Annex F), listing all deviations and additions.
- Any base-language overlap or shadowing is error; core/Annex B rules always have priority and win.

---

4. Change Control & Versioning

- **Amendments:** Changes require standards board review and public notice; at least two releases before removal of any existing feature.
- **Deprecation lifecycle:**
  1. Feature emits deprecation warning (`Dxxxx`) for one major version.
  2. Upgrade/migration suggestions must be present in docs, code headers, and all diagnostics.
  3. After grace period, usage triggers hard compile error.
- **Extensions:** Must be manifest-declared and cannot violate/replace existing core constructs.

---

5. Data Types & Memory Model

5.1 Primitive & Derived Types

Type	Description	Universal	TamaOS	Notes	Since vX.Y
int8/16/32/64	Signed integer	Yes	Yes	Bytes 1/2/4/8	v1.0
uint8/16/32/64	Unsigned integer	Yes	Yes	Bytes 1/2/4/8, modulo wrap	v1.0
byte	Unsigned 8-bit	Yes	Yes	Alias for uint8	v1.0
bool	Boolean	Yes	Yes	0/1, `true`/`false` literals	v1.0
float32	IEEE754 32-bit float	Yes	Yes	IEEE 754, implementations must document rounding, overflow, and subnormals	v1.0

float64	IEEE754 64-bit float	Yes	Yes	See above	v1.0
char	8-bit code unit	Yes	Yes	ASCII-only, use Unicode escapes in string	v1.0
string	Sequence of char	Yes	Yes	See §5.2.3 for layout/handling	v1.0

5.2 Alignment And Packing

- All structure alignments explicit; packing (`#[attribute::packed]`) disables padding only if present.
- Structure field names must be unique; error otherwise (`E0231`).

5.3 Pointers And Array Semantics

- Typed pointers; arithmetic in `freestanding` only.
- **Arrays**:
    - Must be explicitly sized (`type name[size]`)
    - Zero-initialized unless `#[attribute::uninit]` present.
    - Out-of-bounds access is undefined behavior: implementations should warn statically if possible (see diagnostics §3.3.4).
    - Multidimensional arrays: `type name[x][y]`; row-major; bounds enforced at definition.
    - No implicit decay to pointer: explicit cast/conversion required.
- Pointer/array bounds must be enforced where statically resolvable.

5.4 String Handling

- Strings are arrays of `char` with explicit length, not necessarily null-terminated unless marked as C-compatible (`#[attribute::cstring]`).
- Operations (copy, length, comparison) are always explicit.
- Concatenation only via standard library.

5.5 Memory Allocation

5.5.1 alloc/free

- Provided only in `freestanding` or platform-specific extensions; forbidden elsewhere (`E4400`).
- API must be explicit:
    - `void* alloc(size_t bytes);`
    - `void free(void* ptr);`
- Implementations must document stack/heap region used; deterministic in outcome, or error on OOM.
- No implicit allocation performed by compiler or runtime (explicit user request required).

5.5.2 Stack vs Heap

- Stack/heap region boundaries are explicit and implementation-defined; overflow/underflow must trap or abort by default, per platform manifest.

5.5.3 Array Zero-Initialization and UB

- Arrays are zero-initialized by default; uninitialized only with explicit attribute.
- Use of uninitialized memory yields undefined behavior: emit warning at compile if detectable.

---

6. Modularization, Libraries, and File Handling

6.1 Namespaces

- All symbols must be defined within named, top-level namespaces
- Duplicates or ill-formed namespaces: error
- Header exports (`.ghh`) only for interfaces; source in headers forbidden (`E1610`)
- Cyclic or reentrant includes produce fatal `E1540`

6.2 Import/Include Rules

- `use ns.path;` imports symbols/logic; `include "header.ghh";` brings type/interfaces only
- Flag-incompatible imports are error (`E1610`)
- No wildcard/empty/implicit import

6.3 File Structure and Handling

- `.tmg` source: must begin with namespace block
- `.ghh` header: interface/type only, implementation forbidden

6.4 Symbol Lookup and Link Order

- Lookup order: local namespace → explicit parent namespaces → imported namespaces (in import order).
- No fallback, shadowing, or cyclic lookups permitted.
- Symbol re-exports must be marked explicitly for linkage.

---

7. Flags & Attributes

See full normative flag table in **Annex H.**

- No more than one of `#[flag::freestanding]` or `#[flag::user]` per module/namespace (`E1211`)
- All companion attributes (Annex B) are reserved and blocked from redefinition/symbol shadowing
- Correct attribute propagation and retention required at all toolchain stages

8. Library System Enforcement

8.1 Strict Partitioning

- `core::*` import in non-`freestanding` context → error.
- `user::*` import in `freestanding` context → error.

8.2 Symbol Export, Visibility

- Only explicit `export`-flagged symbols appear in linker output.
- No symbol leakage or cross-partition exposure permitted.

---

9. Inline Assembly (Tamassembly)

Tamassembly is Tamago's abstract, target-independent assembly dialect. It provides a portable way to write low-level code in freestanding contexts without tying the source to a single CPU architecture.

Tamassembly is only permitted in `fn` or `proc` declarations marked `#[flag::freestanding]` (directly or inherited from the namespace). In these contexts, the function/procedure body may consist solely of Tamassembly lines instead of high-level `decl_or_stmt` items.

9.1 Syntax and Placement

When Tamassembly mode is active:

- The body is a sequence of zero or more lines, each being one of:
  - Instruction: `mnemonic { operand } ";"`
  - Label: `identifier ":"`
  - Comment: `";" { any text }`
- High-level constructs (`var`, `const`, `if`, `while`, `for`, assignments, `return`, `break`, `continue`, etc.) are forbidden → error `E3011`
- Parsing follows the asm_line production in §3.2.1, but operands and mnemonics are abstract (defined in Annex B).

9.2 Target Selection

The target architecture is determined at build time via an explicit compiler flag (e.g. `--target=armv7m`, `--target=riscv32`, `--target=x86_64`, `--target=aarch64`).

No per-source attributes are used for target selection.

The compiler backend MUST translate each Tamassembly instruction to semantically equivalent native machine code for the selected target. The exact translation sequences are backend-defined but must be documented by the implementation and remain deterministic for a given source + target combination.

If an instruction cannot be mapped efficiently or at all on the selected target → error `E3020` ("Instruction not supported on target <target>").

9.3 Restrictions & Diagnostics

- Tamassembly is forbidden outside `#[flag::freestanding]` contexts → error `E3010`.
- Invalid abstract mnemonic, operand form, or register name → error `E3002` (with reference to Annex B).
- Undefined labels/symbols are linker errors.

---

10. Compiler Output, Standard Library, ABI, Build Config

10.1 Core & Standard Library Signatures

Namespace	Signature	Description	Error & Flag Restrictions
core.memory	fn memcpy(dest: pointer, src: pointer, n: uint32) -> void	Bounded copy	User-mode forbidden
core.memory	fn memset(dest: pointer, value: uint8, n: uint32) -> void	Set memory	User-mode forbidden
core.memory	fn memcmp(a: pointer, b: pointer, n: uint32) -> int32	Compare memory	None
core.memory	fn alloc(n: uint32) -> pointer	Allocate memory, returns null on OOM	Freestanding only (`E4400`)
core.memory	fn free(p: pointer) -> void	Deallocate memory; p must be pointer from alloc; UB if not	Only valid for alloc'd blocks; user/freestanding as per build flags; error on double-free, non-alloc p

core.env	fn env_query(key: string) -> int32	Look up env var, -1 if missing	Must fail gracefully if absent
core.string	fn string_concat(a: string, b: string) -> string	Concatenate strings	No null-termination unless marked
core.string	fn string_copy(dest: string, src: string, max: uint32) -> void	Bounded string copy	

- **Note**
    - `free()` may *only* be called with pointer returned by `alloc()`. Double-free or invalid free triggers diagnostic or undefined behavior.
    - All signatures are strictly enforced; see §11, error codes.
- Build configuration (OS/freestanding, platform/hosted) must be explicit in all builds.

---

11. Semantics & Contracts

11.1 Function, Procedure, and Operator Semantics

11.1.1 Operators: List, Precedence, Casts

Level	Operators	Assoc.	Description
1	cast(expr)	right	Type cast (explicit)
2	`*`, `-`, `!`, `~`, `+` (unary)	right	Unary ops
3	`*`, `/`, `%`	left	Multiplicative
4	`+`, `-`	left	Additive
5	`<<`, `>>`	left	Bit shifts
6	`&`	left	Bitwise AND
7	`^`	left	Bitwise XOR
8	`	`	left
9	`<`, `<=`, `>`, `>=`	left	Comparison
10	`==`, `!=`	left	Equality
11	`&&`	left	Logical AND
12	`		`

13	`=`, `+=`, `-=`, `*=`, `/=`, etc.	right	Assignment

- **Ternary ('?:') remains strictly banned and triggers parse error.**
- **All binary/unary operators are listed here and reflected in §3.2.1 grammar.**
- **No implicit type conversions.**
- Operator corner cases and type safety are enforced, including bitwise support for `'|'`.

11.1.2 Identifier and Naming Rules

- Identifier rules are as stated in grammar:
    - Must start with `_` or alpha, may not start with digit or double underscore.
    - `__` as leading chars forbidden for user code: triggers `E0260`.
    - No Unicode in identifiers.
- **All identifier usage is grammar-validated; see examples and diagnostic codes.**

11.1.3 Functions, Procedures, and Variadics

- `fn` = function, value-returning; `proc` = procedure, void-returning.
- Parameter and arity rules as per grammar and checklist.
- **Variadic notation/attribute only permitted as last parameter (with attribute, enforced by grammar).**
- **No nested functions, lambdas, or closure semantics (non-goal; forbidden syntax).**
- **Entry Points:** Only explicit `fn main(argc: int32, argv: pointer) -> int32` (user mode) or `fn start() -> void` (freestanding).
- **All declaration forms, operator usage, and contracts are EBNF-enforced.**

---

12. Annexes

Annex A. Rationale (Informative)

All critical design choices—determinism, partitioned flagging, absolute layout/type explicitness, and prohibition of reflection/dynamic behaviors—are justified by the need for high-assurance, traceable, platform-independent code suitable for regulated and embedded environments. Compiler-verified modularity and flag enforcement prevent privilege escalation and inadvertent unsafe features found in legacy or scripting-centric languages. **Tamassembly** was introduced in v1.0 to reduce ceremony in the primary freestanding use-case (bootloaders, firmware, microkernels) while preserving strict mode separation, deterministic builds, and comprehensive diagnostics. By making it the only inline assembly form, the language avoids dual syntax maintenance and encourages explicit freestanding intent.

Annex B. Reserved Words, Opcodes, and Grammar (Normative)

Full EBNF (ISO-style) cross-listed from §3.2.1. Operator/cast rules and precedence table included. Ternary operator is a non-goal and causes parse error if used. All grammar productions here are normative.

B.1 Reserved Words, Keywords, and Built-in Elements (Normative)

This subsection lists all tokens that are reserved or have special meaning in the core language (outside of Tamassembly).

B.1.1 Reserved Keywords (Normative)

The following tokens are reserved and cannot be used as identifiers: namespace, use, include, `var`, `const`, `fn`, `proc`, `struct`, `enum`, `if`, `else`, `while`, `for`, `break`, `continue`, `return`, `cast`, `true`, `false`, `void`

B.1.2 Built-in Type Names (Normative)

These are reserved in type contexts and cannot be redefined: `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, `uint64`, `float32`, `float64`, `char`, `bool`, `string`, `pointer`

B.1.3 Other Reserved Tokens (Normative)

The following are reserved for future use or special syntax: asm (deprecated), attribute, flag

Any use of these as identifiers triggers error `E0001` (see §3.1.2).

B.2 Tamassembly (Normative)

Tamassembly is Tamago's abstract, target-independent assembly dialect. It provides a portable way to write low-level code in freestanding contexts without tying the source to a single CPU architecture.

The compiler backend translates each Tamassembly instruction to one or more native instructions for the selected target architecture (determined at build time via `--target` flag, e.g. `--target=armv7m`, `--target=riscv64`, `--target=x86_64`, `--target=aarch64`).

Translation sequences are backend-defined but must be semantically equivalent, deterministic for the same source + target combination, and documented by the implementation.

B.2.1 Symbolic / Virtual Registers (Normative)

Registers are symbolic and untyped. The backend performs register allocation, spilling to stack, or temporary materialization as needed.

Defined registers (always available):

- General-purpose: `r0`, `r1`, `r2`, ..., `r31` (at least 32 available; backends may support more via virtual registers or spilling)
- Special-purpose:
  - `sp` – stack pointer
  - `lr` – link register (return address)
  - `pc` – program counter (read-only; usable only in PC-relative addressing)
  - flags – abstract condition flags (zero, negative, carry, overflow, sign, parity)

B.2.2 Operands Valid operand forms (case-insensitive, Normative):

- Register: `r5`, `sp`, `lr`, `pc`, flags
- Immediate: `#42`, `#0x1000`, `#-8`, `#0b101010`
- Label reference: `my_loop`, `data_table`
- Memory access:
  - `[base]`
  - `[base + #offset]`
  - `[base + index]`
  - `[base + index * scale] (scale = 1, 2, 4, 8)`
  - `[base + index * scale + #offset]`
- PC-relative: label(pc) (for literals or data)

B.2.3 Abstract Mnemonics (Normative List)

All arithmetic, logic, and some other instructions support an optional {s} suffix to update condition flags. All instructions support an optional {cond} suffix (see B.2.4) for conditional execution.

Move / Set

```
mov dst, src          ; dst ← src (register or label address)

mov dst, #imm         ; dst ← immediate

movz dst, #imm        ; dst ← zero-extended immediate

movn dst, #imm        ; dst ← inverted immediate
```

Arithmetic

```
add dst, src1, src2/#imm

sub dst, src1, src2/#imm

mul dst, src1, src2          ; signed
```

```
umul dst, src1, src2       ; unsigned
div dst, src1, src2        ; signed
udiv dst, src1, src2       ; unsigned
rem dst, src1, src2        ; signed remainder
urem dst, src1, src2       ; unsigned remainder
```

Bitwise

```
and dst, src1, src2/#imm
or dst, src1, src2/#imm
xor dst, src1, src2/#imm
not dst, src               ; dst ← ~src
bic dst, src1, src2        ; dst ← src1 & ~src2
shl dst, src, #imm/reg           ; logical left
shr dst, src, #imm/reg           ; logical right
sar dst, src, #imm/reg           ; arithmetic right
rol dst, src, #imm/reg           ; rotate left
ror dst, src, #imm/reg           ; rotate right
```

Compare / Test (set flags)

```
cmp a, b/#imm         ; a – b
cmn a, b/#imm         ; a + b
test a, b/#imm        ; a & b
tst a, #imm           ; alias for test a, #imm
```

Load / Store

```
ldr dst, [base{, #offset}]
ldr dst, [base, index{, *scale}]
ldr dst, [base + index * scale + #offset]
ldr dst, label(pc)               ; PC-relative
str src, [base{, #offset}]
ldrb dst, [base{, #offset}]      ; 8-bit load
ldrh dst, [base{, #offset}]      ; 16-bit load
```

`strb src, [base{, #offset}]` ; 8-bit store

`strh src, [base{, #offset}]` ; 16-bit store

Branch / Call

`b label` ; unconditional

`bl label` ; branch and link

`br reg` ; branch to register

`ret` ; pc ← lr

Conditional branches (using abstract flags)

`beq` / `bne` / `bgt` / `bge` / `blt` / `ble` / `bhi` / `blo` / `bhs` / `bls` label

System / Control

`syscall #num` ; system call

`nop` ; no operation

`break` ; debug trap

`wfi` ; wait for interrupt (optional)

Pseudo-operations (directives)

`.global identifier`

`.equ identifier, value`

`.byte value{, value...}`

`.word value{, value...}`

`.align n` ; 2^n bytes

`.section name`

B.2.4 Condition Suffixes {cond} (Normative)

`eq` equal / zero

`ne` not equal

`gt` greater than (signed)

`ge` greater or equal (signed)

`lt` less than (signed)

`le` less or equal (signed)

`hi` higher (unsigned)

`hs` higher or same (unsigned)

`lo` lower (unsigned)

`ls` lower or same (unsigned)

`mi` minus / negative

`pl` plus / non-negative

`vs` overflow set

`vc` overflow clear

Backend MUST map to target-appropriate conditional mechanisms.

B.2.5 Backend Responsibilities (Normative)

- Produce semantically equivalent code.
- Translation may emit multiple native instructions.
- Follow target ABI for registers, stack, calling convention.
- Deterministic output for same source + target.
- Document supported targets and limitations.
- Error `E3020` if instruction cannot be mapped.

B.2.6 Recommended Targets (Informative)

- `armv7m`
- `riscv32imac`
- `riscv64imac`
- `x86_64`
- `aarch64`

Annex C. Conformance Checklist (Normative)

#	Requirement
1	Namespace/import/include: All declarations must be scoped, no wildcards/alias permitted.
2	Identifier: No starting with digit or `__`; no Unicode in identifiers; enforce E0260.
3	All productions tested: Declarations, expressions, control flow, statement forms.
4	Explicit attribute positions for flags/variants; must precede item or parameter.
5	Function and procedure forms: `fn` returns value, `proc` returns void.

6	Variadic: Only permitted as last parameter, with explicit attribute.
7	No nested functions, lambdas, or closures (tested by forbidden code, EBNF violation).
8	Operator table: Full coverage of precedence, associativity; all uses represented.
9	`cast<T>(expr)` syntax is required everywhere a cast is used.
10	String handling: assignment, context, escapes, UTF-8 encoding for `\uXXXX`.
11	CString support: Null-terminated only via attribute.
12	Standard library signatures: Conformance with core.memory and core.string, errors handled.
13	Inline asm: Attributes, mnemonics, and manifest.
14	Diagnostic errors for invalid identifiers, forbidden constructs, operator misuse.
15	Pointer use: only via allowed means; address-of (&) if and only if present in core.
16	Free/memory lifetime: Double-free or invalid free must trigger error.
17	Entry point: Only explicit `fn main(argc: int32, argv: pointer) -> int32` or `fn start() -> void`.
18	No wildcard or as/alias in import paths.
19	Array, struct, enum, pointer declaration and initialization forms must conform.
20	All grammar and semantic rules validated by real code (see Annex I).

Annex D. ABI/Entry/Linking Conventions (Normative)

- **Call convention:** args left-to-right in registers, ret in R0
- **Entry points:** program must provide either `main` or `start`
- **Segments:** declared explicitly; mismatch triggers error
- **Linking:** only explicitly exported symbols can be linked

Annex E. Diagnostic Codes & Examples (Normative)

Code	Violation	Example Output

E0001	Keyword as identifier	error[E0001]: reserved keyword ...
E0230	Packed attribute conflict / invalid packing	error[E0230]: packed attribute conflict / invalid packing
E0231	Duplicate structure field name	error[E0231]: duplicate structure field name
E2001	Declaration outside namespace	error[E2001]: declaration outside namespace ...
E0260	Double-underscore user identifier	error[E0260]: identifier '__foo' reserved for implementation
E1211	Multiple conflicting flags on namespace/module	error[E1211]: multiple conflicting flags on namespace/module
E1241	Invalid use of cstring attribute	error[E1241]: invalid use of cstring attribute
E1281	Invalid variadic usage	error[E1281]: variadic only as last function parameter
E1283	Implicit type conversion	error[E1283]: no implicit promotion allowed
E5001	Forbidden cast (e.g., int to string)	error[E5001]: illegal cast; value/type conversion forbidden
E2114	Function call arity mismatch	error[E2114]: function 'foo' with 1 argument (expected 2)
E4400	alloc in user mode	error[E4400]: allocation forbidden in user mode
E4002	Double free / invalid free	error[E4002]: invalid or double free
E3020	Instruction not supported on target	error[E3029]: instruction not supported on target
E3010	Tamassembly used outside freestanding context	error[E3010]: tamassembly used outside freestanding context
E3011	High-level construct inside Tamassembly body	error[E3011]: high-level construct inside Tamassembly body
E3002	Invalid mnemonic/operand/register	error[E3002]: invalid mnemonic/operand/register

- Color/highlighting is recommended but not required.
- All user-facing diagnostics must be in UTF-8 (i18n normative in v1.0).

Annex F. Internationalization, Security, Portability (Informative)

- **Internationalization:** Error codes must be language-neutral, allowing user-facing messages in any supported locale.
- **Security:** Dynamic privilege escalation, hidden dependencies, and runtime indeterminism are strictly forbidden.
- **Portability:** All alignment, type sizing, and binary layout must be rigorously explicit. Multi-target toolchains must register a compatibility manifest.

Annex G. Diagnostic Classes & Convention (Normative)

G.1 Diagnostic Classes (Normative)

- **error** (Exxxx) Hard failure — compilation stops immediately. The program is invalid and cannot be built. Must be fixed.
- **warning** (Wxxxx) Potential issue or bad practice — compilation continues, but the code may have problems (e.g. undefined behavior risk, inefficiency, style issues). Should be fixed.
- **hint** (Hxxxx) Non-blocking informational note or suggestion — provides alternative ways to write code, style recommendations, or minor improvements. Optional to follow.
- **deprec** (Dxxxx) Deprecated feature or syntax — compilation continues (usually as warning), but the item will become a hard error in a future major version. Migration path must be provided.

G.2 Code Ranges (Normative)

All codes are 4–5 digits long and grouped by category for clarity and future expansion:

- **E0xxx** (0000–0999) — Lexical, syntax, grammar, and parsing errors
- **E1xxx** (1000–1999) — Type system, semantic, and type-checking errors
- **E2xxx** (2000–2999) — Declaration, scope, visibility, and name resolution issues
- **E3xxx** (3000–3999) — Tamassembly and inline assembly errors
- **E4xxx** (4000–4999) — Memory, allocation, lifetime, and ownership issues
- **E5xxx** (5000–5999) — Cast, conversion, operator, and expression misuse
- **E6xxx** (6000–6999) — Import, include, module, and namespace errors
- **E7xxx** (7000–7999) — Flag, attribute, privilege, and security violations
- **E8xxx** (8000–8999) — Linkage, ABI, entry point, and build configuration issues

- **E9xxx** (9000–9999) — Reserved for target-specific, backend, extension, or future errors
- **Wxxxx** — Warnings (any 4–5 digit number starting with W)
- W0xxx–W1xxx: Style, formatting, and portability warnings
- W2xxx–W9xxx: Performance, best-practice, and potential UB warnings
- **Hxxxx** — Hints (any 4–5 digit number starting with H)
- Used for non-blocking suggestions (e.g. H1001: "consider using const here")
- **Dxxxx** — Deprecations (any 4–5 digit number starting with D)
- Usually emitted as warnings; become errors in future major versions

G.3 Reserved Ranges for Future Expansion (Normative)

- E9000–E9999: Reserved specifically for target/backend/extension errors
- W2000–W2999: Reserved for performance/style warnings
- Hxxxx: Open range for hints (no strict sub-ranges)
- Dxxxx: Open range for deprecations

Annex H. Flags and Attributes Catalog (Normative)

Catalog Table

Name	Scope	Effect	Error on Conflict	Inheritance/Override Rules	Since vX.Y
flag::freestanding	namespace	Enables privileged ops	Yes (E1211)	Not inherited; must be lexically specified	v1.0
flag::user	namespace	Restricts to user-mode only	Yes (E1211)	Not inherited; see §7	v1.0
attribute::packed	struct	Disable padding	Yes (E0230)	Inherits unless overridden at field/member	v1.0
attribute::cstring	string/var	Null-terminated string	Yes (E1241)	Shadows local attribute	v1.0
attribute::variadic	function	Allow variadic args	Yes (E1281)	Must be top-level function, not closure	v1.0

- **Conflict**: any conflicting flag/attribute errors; fix suggestion required.

Annex I. Conformance Tests and Reference Examples (Normative)

- Legal and illegal samples for:
    - Syntax (all productions in EBNF)
    - Array, pointer, and struct zero-init, allocation
    - Operator precedence and overflow
    - Import, flags, asm/ISA
    - Diagnostics and contract checks

Example: Usage of Tamassembly in Tamago

```
#[flag::freestanding]

proc start() -> void {

    mov     sp, #stack_top;

    bl      init;

loop:

    nop;

    b       loop;

}
```

Annex J. Standards Compliance (Informative)

This annex lists the ISO/IEC standards referenced by this specification. Some standards provide guidance for design intent or quality goals and are not mandatory for conformance, while others are required to implement or interpret the language correctly.

Annex J.1 Informative standards (guidance / design intent)

These standards are used as guidance for design intent or quality goals and are **not mandatory for conformance**:

- **ISO/IEC 25010** - Systems and software Quality Requirements and Evaluation (SQuaRE) — maps language features to quality characteristics (reliability, maintainability, etc.).
- **ISO 639-1 / ISO 639-2** — Language codes (identifies the natural language of the specification).
- **ISO 8601** — Date and time representation, if timestamps are defined.
- **ISO 19005 (PDF/A)** — Long-term archival format of the document.

Annex J.2 Normative standards (required for implementation / interpretation)

These standards are **required to implement or interpret the language correctly**:

- **ISO/IEC Directives, Part 2** — Rules for the structure and drafting of international standards.
- **ISO/IEC 10646** — Universal Coded Character Set (Unicode) for source text and identifiers.
- **ISO/IEC/IEEE 60559** — Floating-point arithmetic, if applicable.
- **ISO/IEC 10967** — Language-independent arithmetic, if applicable.

Glossary (Normative)

- **Flag:** Compile-time scope control, e.g., `freestanding`.
- **Undefined behavior (UB):** Action outside language rules; must be diagnosed if statically detectable.
- **Zero-initialization:** Memory set to all-bits-zero (not always logical zero for float/ptr types).
- **Attribute:** Metadata (e.g., contract, cstring). Always placed before the keyword/parameter.
- **Cast:** Explicit type conversion, required for all conversions.
- **CString:** `string` with `#[attribute::cstring]`: null-terminated, interop-use.
- **EBNF:** Extended Backus–Naur Form, full grammar in §3.2.1 and Annex B.
- **Namespace:** Code module, all declarations enclosed, enforced by grammar.
- **Operator:** All symbols, precedence/fixity in §11.1.1/Annex B.
- **Proc:** Procedure, strictly `void` (side effects only); no return value.
- **Struct/Enum:** Type definitions, see grammar for syntax.
- **Variadic:** Final function parameter only, explicit attribute. See EBNF/error `E1281`

Index (Normative)

All additions/changes are clearly marked. This document is now deemed full feature-complete for parser/reference implementation. All items intended for implementer review (marked "placeholder") must be explicitly reviewed in committee or by implementer prior to ratification.

End of Tamago Language Official Specification v1.0

Authored by 陳藤原（ふじわら・りん；　**Fujiwara Rin**）

Github: @fujiwararin1303

2026, February 1（２０２５年２月１日）